

# L-Store: A Real-time OLTP and OLAP System

Mohammad Sadoghi<sup>†</sup>, Souvik Bhattacharjee<sup>‡\*</sup>, Bishwaranjan Bhattacharjee<sup>†</sup>, Mustafa Canim<sup>†</sup>

<sup>†</sup>IBM T.J. Watson Research Center

<sup>‡</sup>University of Maryland, College Park

## ABSTRACT

Arguably data is a new natural resource in the enterprise world with an unprecedented degree of proliferation. But to derive real-time actionable insights from the data, it is important to bridge the gap between managing the data that is being updated at a high velocity (i.e., OLTP) and analyzing a large volume of data (i.e., OLAP). However, there has been a divide where specialized solutions were often deployed to support either OLTP or OLAP workloads but not both; thus, limiting the analysis to stale and possibly irrelevant data. In this paper, we present Lineage-based Data Store (L-Store) that combines the real-time processing of transactional and analytical workloads within a single unified engine by introducing a novel lineage-based storage architecture. We develop a contention-free and lazy staging of columnar data from a write-optimized form (suitable for OLTP) into a read-optimized form (suitable for OLAP) in a transactionally consistent approach that also supports querying and retaining the current and historic data. Our working prototype of L-Store demonstrates its superiority compared to state-of-the-art approaches under a comprehensive experimental evaluation.

## 1. INTRODUCTION

We are now witnessing an architectural shift and divide in database community. The first school of thought emerged from an academic conjecture that “*one size does not fit all*” [35] (i.e., *advocating specialized solutions*), which has led to manifolds of innovations over the last decade in creating specialized database engines geared toward various niche workloads and application scenarios (e.g., [35, 5, 11, 9, 28, 8, 21, 36]). This school has successfully influenced major database vendors such as Microsoft to focus on building new specialized engines offered as loosely integrated engines (e.g., Hekaton in-memory engine [8] and Apollo column store engine [19]) within a single umbrella of database portfolio (notably, recent efforts are now focused on a tighter real-time integration of Hekaton and Apollo engines [18]). It has also influenced Oracle to partially accept the basic premise that “one size does not fit all” as far as data representation is concerned and has led Oracle to develop a dual-format technique [16] that maintains two tightly integrated representation of data (i.e., two copies of the data) in a transactionally consistent manner.

However, the second school of thought, supported by both academia (e.g., [13, 6, 7]) and industry (e.g., SAP [10]), rejects the aforementioned fundamental premise and advocates a generalized solution. Proponents of this idea, rightly in our view, make the following arguments. First, there is a tremendous cost in building and maintaining multiple engines from both the perspective of database

vendors and users of the systems (e.g., application development and deployment costs). Second, there is a compelling case to support real-time decision making on the latest version of the data [27] (likewise supported by [16, 18]), which may not be feasible across loosely integrated engines that are connected through the extract-transform load (ETL) process. Closing this gap may be possible, but its elimination may not be feasible without solving the original problem of unifying OLTP and OLAP capabilities or without being forced to rely on ad-hoc approaches to bridge the gap in hindsight. We argue that the separation of OLTP and OLAP capabilities is a step backward that defers solving the actual challenge of real-time analytics. Third, combining real-time OLTP and OLAP functionalities remains as an important basic research question, which demands deeper investigation even if it is purely from the theoretical standpoint.

In this dilemma, we support the latter school of thought (i.e., *advocating a generalized solution*) with the goal of undertaking an important step to study the entire landscape of single engine architectures and to support both transactional and analytical workloads holistically (i.e., “*one size fits all*”). In this paper, we present Lineage-based Data Store (L-Store) with a novel lineage-based storage architecture to address the conflicts between row- and column-major representation by developing a contention-free and lazy staging of columnar data from write optimized into read optimized form in a transactionally consistent manner without the need to replicate data, to maintain multiple representation of data, or to develop multiple loosely integrated engines that sacrifices real-time capabilities.

To further disambiguate our notion of “*one size fits all*”, in this paper, we restrict our focus to real-time relational OLTP and OLAP capabilities. We define a set of architectural characteristics for distinguishing the differences between existing techniques. First, there could be a single product consisting of multiple loosely integrated engines that can be deployed and configured to support either OLTP or OLAP. Second, there could be a single engine as opposed to having multiple specialized engines packaged in a single product. Third, even if we have a single engine, then we could have multiple instances running over a single engine, where one instance is dedicated and configured for OLTP workloads while another instance is optimized for OLAP workloads, in which the different instances are assumed to be connected using an ETL process. Finally, even when using the same engine running a single instance, there could be multiple copies or representations (e.g., row vs. columnar layout) of the data, where one copy (or representation) of the data is read optimized while the second copy (or representation) is write optimized. The architectural comparison of various existing techniques based on our rigorous definition of

\*Work was performed as part of a summer internship at IBM T.J. Watson Research Center under Mohammad Sadoghi’s mentorship.

	L-Store	HANA [27]	ES2 [6, 7]	HyPer [13]	Oracle Dual-format [16]	Microsoft SQL Server [18]	H-Store+Hadoop [11]
Single Product	✓	✓	✓	✓	✓	✓	–
Single Engine	✓	✓	✓	✓	✓	–	–
Single Instance	✓	✓	✓	✓	✓	✓	–
Single Copy	✓	✓	✓	data page replication (OS forking)	–	–	–
Single Representation	✓	main + delta	✓	✓	–	–	–
OLTP-optimized	✓	✓	limited to get/put operations	limited to partitionable workloads (i.e., serial execution)	✓	✓	✓
OLAP-optimized	✓	✓	inconsistent snapshot is possible	✓	✓	✓	✓
Unified OLTP+OLAP	✓	✓	✓	✓	✓	✓	–

Table 1: Architectural characterization of selected database engines.

“one size fits all” is outlined in Table 1.<sup>1</sup>

In short, we develop L-Store, as an important first step towards supporting real-time OLTP and OLAP processing that faithfully satisfies our definition of *generalized solution*, and, in particular, we make the following contributions:

- Introducing a contention-free update mechanism over a native columnar storage model in order to lazily and independently stage stable data from a write-optimized columnar layout (i.e., OLTP) into a read-optimized columnar layout (i.e., OLAP)
- Achieving (at most) 2-hop away access to the latest version of any record (preventing read performance deterioration for point queries)
- Contention-free merging of only stable data, namely, merging of the read-only base data with recently committed updates (both in columnar representation) without the need to block ongoing or new transactions
- Contention-free page de-allocation (upon the completion of the merge process) using an epoch-based approach without the need to drain the ongoing transactions
- A first of its kind comprehensive evaluation to study the leading architectural design for concurrently supporting short update transactions and analytical queries (e.g., an in-place update with a history table architecture and the commonly employed main and delta stores architecture)

## 1.1 Motivating Real-time OLTP and OLAP

Before describing our proposed approach in-depth, we briefly present two important scenarios that benefit greatly from a real-time OLTP and OLAP solution.

Consider the mobile e-commerce market, in which the revenue for the location-based mobile advertising alone is expected to reach \$18 billions by 2019 [25]. A potential buyer with a mobile device may roam around physically while shopping. In the meantime, the shopper’s mobile device generates location information. Alternatively, as the shopper browses the web, again the location information is either exchanged explicitly or detected automatically based on the shopper’s IP address or by its connection to the nearby WiFi

<sup>1</sup>However, it is crucial to note that the presented comparison is solely focused on the overall architectural choices, and it does not make any claims about the relative system performance and/or functionalities. For example, if HANA contains more check marks than Microsoft SQL Server, it does not imply that HANA is a better product, instead it simply assesses HANA architecturally with respect to our definition of “one size fits all”.

routers. Now the task of any real-time targeted advertising auction is to determine and present a set of relevant ads to the shopper by running analytics over the location information, shopping patterns, past purchases, and browsing history of the shopper. Furthermore, if these advertisements result in a purchase, then the resulting transactions need to become available immediately to subsequent analytics in order to improve the effectiveness of future advertisements. Moreover, the actual ad bidding in the auction also requires a transactional semantics support in real-time. Finally, all these steps must be completed typically within 150 milliseconds [2]. Therefore, we argue that in order to sustain a high velocity transactional data (e.g., Google AdWords served almost 30 billion ads per day in 2012 [14]) while executing complex analytics on the latest and historic (transactional) data, there is a compelling need to develop a solution that exhibits a true real-time OLTP and OLAP capabilities.

Another prominent scenario is fraud detection especially at the time when the cost of cybercrime continues to increase at a staggering rate and has already surpassed \$400 billion dollars annually [1]. For instance, a credit card company will need to approve a transaction in a small time window (i.e., subsecond ranges). During this short time span, it is forced to determine if a transaction is fraudulent or not. Thus, there is a crucial need to run complex analytics in real-time as part of the transaction that is being processed. Without such a proactive fraud detection capability, fraudulent transactions may remain undetectable, which may result in irreversible financial losses as clearly been witnessed when billions of dollars are being lost due to fraud activities every year [1]. Furthermore, there are indirect financial losses involving stakeholders such as credit card companies and merchants. The indirect losses attributed to decline of legitimate transactions that disrupts merchant’s daily operation, lost payment volume as consumers opt for alternative payment types that are perceived to be safer, and lost customers due to card cancellation and reissue [26].

## 2. UNIFIED ARCHITECTURE

The divide in the database community is partly attributed to the storage conflict pertaining to the representation of transactional and analytical data. In particular, transactional data requires write-optimized storage, namely the row-based layout, in which all columns are co-located (and preferably uncompressed for in-place updates). This layout improves point update mechanisms, since accessing all columns of a record can be achieved by a single I/O (or few cache misses for memory-resident data). In contrast, to optimize the analytical workloads (i.e., reading many records), it is important to have read-optimized storage, i.e., columnar layout in highly compressed form. The intuition behind having columnar layout is due to the observation that most analytical queries tend to access only a small subset of all columns [3]. Thus, by storing data column-wise,

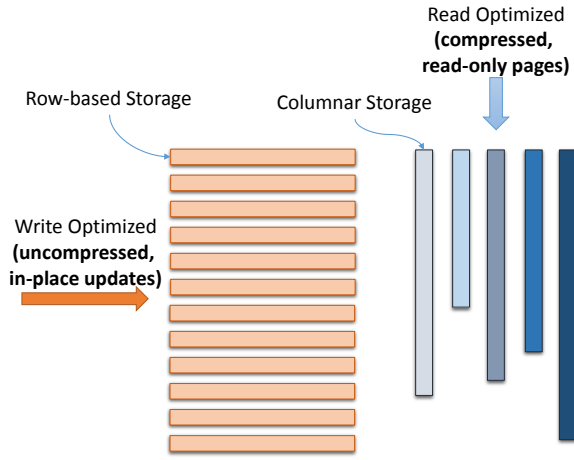


Figure 1: Overview of storage layout conflict.

we can avoid reading irrelevant columns (i.e., reducing the raw amount of data read) and avoid polluting processor’s cache with irrelevant data, which substantially improve both disk and memory bandwidth, respectively. Furthermore, storing data in columnar form improves the data homogeneity within each page, which results in an overall better compression ratio. This storage conflict is depicted in Figure 1.

## 2.1 L-Store Storage Overview

To address the dilemma between write- and read-optimized layouts, we develop L-Store. As demonstrated in Figure 2, the high-level architecture of L-Store is based on native columnar layout (i.e., data across columns are aligned to allow implicit re-construction), where records are (virtually) partitioned into disjoint ranges (also referred to as update range). Records within each range span a set of read-only, compressed pages, which we refer to them as the *base pages*. More importantly, for every range of records, and for each updated column within the range, we maintain a set of append-only pages to store the latest updates, which we refer to them as the *tail pages*. Anytime a record is updated in base pages, a new record is appended to its corresponding tail pages, where there are explicit values only for the updated columns (non-updated columns are preassigned a special null value when a page is first allocated). We refer to the records in base pages as the *base records* and the records in tail pages as the *tail records*. Each record (whether falls in base or tail pages) spans over a set of aligned columns (i.e., no join is necessary to pull together all columns of the same record).<sup>2</sup>

A unique feature of our lineage-based architecture is that tail pages are strictly append-only and follow a write-once policy. In other words, once a value is written to tail pages, it will not be over-written even if the writing transaction aborts. The append-only design substantially simplifies concurrency and recovery protocol as described in Section 4.1. Another important property of our lineage-based storage is that all data are represented in a common holistic form; there are no ad-hoc corner cases. Records in both base and tail pages are assigned record-identifiers (RIDs) from the same key space. Therefore, both base and tail pages are referenced through the database page directory using RIDs and persisted identically. Therefore, at the lower-level of the database stack, there is absolutely no difference between base vs. tail pages or base vs. tail

<sup>2</sup>Fundamentally, there is no difference between base vs. tail record, the distinction is made only to ease the exposition.

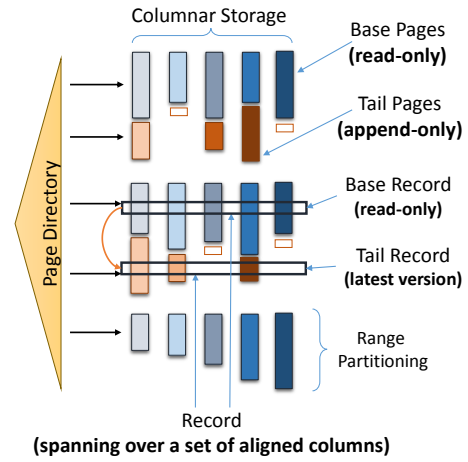


Figure 2: Overview of the lineage-based storage architecture.

records; they are presented and maintained identically.

To speed query processing, there is also an explicit linkage (forward and backward pointers) among records. From a base record, there is a forward pointer to the latest version of the record in tail pages. The different versions of the same records in tail pages are chained together to enable fast access to an earlier version of the record. The linkage is established by introducing a table-embedded indirection column that stores forward pointers (i.e., RIDs) for base records and backward pointers for tail records (i.e., RIDs).

The final aspect of our lineage-based architecture is a periodic, contention-free merging of a set of base pages with its corresponding tail pages. This is performed to consolidate base pages with the recent updates and to bring base pages forward in time (i.e., creating a set of merged pages). Tail pages that are already merged and fall outside the snapshot boundaries of all active queries are called historic tail-pages. These pages are re-organized, so that different versions of a record are stored contiguously inlined. Delta-compression is applied across different versions of tail records, and tail records are ordered based on the RIDs of their corresponding base records. Below, we describe the unique design and algorithmic features of L-Store that enables efficient transactional processing without performance deterioration of analytical processing; thereby, achieving a real-time OLTP and OLAP.

## 2.2 Lineage-based Storage Architecture

In L-Store, the storage layout is naively columnar that applies equally to both base and tail pages. A detailed view of our lineage-based storage architecture is presented in Figure 3. In general, one can perceive tail pages as directly mirroring the structure and the schema of base pages. As we pointed out earlier, conceptually for every record, we distinguish between base vs. tail records, where each record is assigned a unique RID. But it is important to note that the RID assigned to a base record is stable and remains constant throughout the entire life-cycle of a record, and all indexes only reference base records (base RIDs); consequently, eliminating index maintenance problem associated when update operation results in creation of a new version of the record [33, 32]. A reader performing index lookup always lands at a base record, and from the base record it can reach any desired version of the record by following the table-embedded indirection to access the latest (if the base record is out-of-date) or an earlier version of the record. However, when a record is updated, a new version is created. Thus, a new tail record is created to hold the new version, and the new

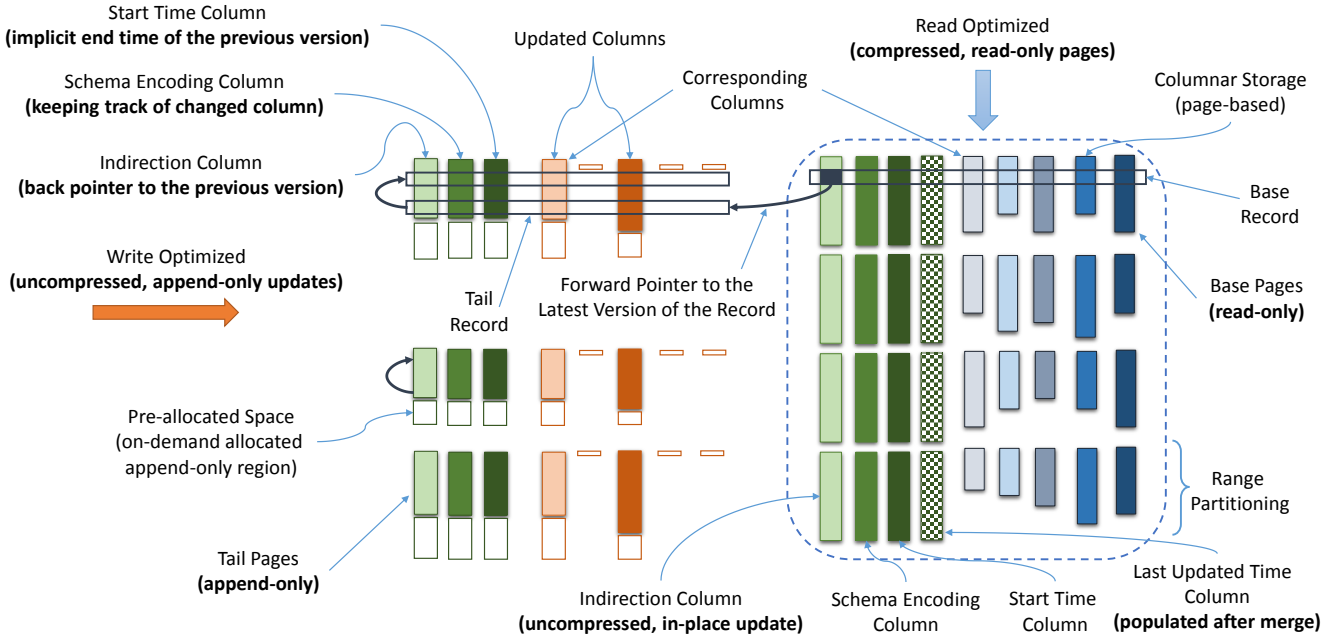


Figure 3: Detailed, unfolded view of lineage-based storage architecture.

tail record is assigned a new tail RID that is referenced by the base record (as demonstrated in Figure 3).

Each table in addition to having the standard data columns has several meta-data columns. These meta-data columns include the *Indirection* column, the *Schema Encoding* column, the *Start Time* column, and the *Last Updated Time* column. An example of table schema is shown in Table 2.

The *Indirection* column exists in both the base and tail records. For base records, the *Indirection* column is interpreted as a forward pointer to the latest version of a record residing in tail pages, essentially storing the RID of the latest version of a record. If a record has never been updated, then the *Indirection* column will hold a null value. In contrast, for tail records, the *Indirection* column is used to store a backward pointer to the last updated version of a record in tail pages. If no earlier version exists, then the *Indirection* column will point to the RID of the base record.

The *Schema Encoding* column stores the bitmap representation of the state of the data columns for each record, where there is one bit assigned for every column in the schema (excluding the meta-data columns), and if a column is updated, its corresponding bit in the *Schema Encoding* column is set to 1, otherwise is set to 0. The schema encoding enables to quickly determine if a column has ever been updated or not (for base records) or to determine for each tail record, which columns have been updated and have explicit values as opposed to those columns that have not been updated and have an implicit special null values (denoted by  $\emptyset$ ). An example of *Schema Encoding* column is provided in Table 2.

The *Start Time* column stores the time at which a base record was first installed in base pages (the original insertion time), and for a tail record, the *Start Time* column holds the time at which the record was updated, which is also the implicit end time of the previous version of the record. In addition, to the *Start Time* column, for base records, we maintain an optional *Last Updated Time* column, which is only populated after the merge process is taken place and holds the *Start Time* of those tail records included in merged pages. Also note that the initial *Start Time* column for base records is al-

ways preserved (even after the merge process) for faster pruning of those records that are not visible to readers because they fall outside the reader’s snapshot. Lastly, we may add the *Base RID* column optionally to tail records to store the RIDs of their corresponding base records; this is utilized to improve the merge process. *Base RID* is a highly compressible column that would require at most two bytes when restricting the range partitioning of records to  $2^{16}$  records.

## 2.3 Fine-grained Storage Manipulation

The transaction processing can be viewed as two major challenges: (1) how data is physically manipulated at the storage layer and how changes are propagated to indexes and (2) how multiple transactions (where each transaction consists of many statements) can concurrently coordinate reading and writing of the shared data. The focus of this paper is on the former challenge, and we defer the latter to our discussion on the employed concurrency model in Section 4.1.

### 2.3.1 Update and Delete Procedures

Without the loss of generality, we focus on how to handle a single point update or delete in L-Store (but note that we support multi-statement transactions as demonstrated by our evaluation). Each update may affect a single or multiple records. Since records are (virtually) partitioned into a set of disjoint ranges (as shown in Table 2), each updated record naturally falls within only one range. Now for each range of records, upon the first update to that range, a set of tail pages are created (and persisted on disk optionally) for the updated columns and are added to the page directory, i.e., lazy tail-page allocation. Consequently, updates for each record range are appended to their corresponding tail pages of the updated columns only; thereby, avoiding in-place updates for the data columns and clustering updates for a range of records within their corresponding tail pages.

To describe the update procedure in L-Store, we rely on our running example shown in Table 2. When a transaction updates any column of a record for the first time, two new tail records (each

RID	Indirection	Schema Encoding	Start Time	Key	A	B	C
Partitioned base records for the key range of $k_1$ to $k_3$							
$b_1$	$t_8$	0000	10:02	$k_1$	$a_1$	$b_1$	$c_1$
$b_2$	$t_5$	0101	13:04	$k_2$	$a_2$	$b_2$	$c_2$
$b_3$	$t_7$	0001	15:05	$k_3$	$a_3$	$b_3$	$c_3$
Partitioned base records for the key range of $k_4$ to $k_6$							
$b_4$	$\perp$	0000	16:20	$k_4$	$a_4$	$b_4$	$c_4$
$b_5$	$\perp$	0000	17:21	$k_5$	$a_5$	$b_5$	$c_5$
$b_6$	$\perp$	0000	18:02	$k_6$	$a_6$	$b_6$	$c_6$
Partitioned tail records for the key range of $k_1$ to $k_3$							
$t_1$	$b_2$	0100*	13:04	$\emptyset$	$a_2$	$\emptyset$	$\emptyset$
$t_2$	$t_1$	0100	19:21	$\emptyset$	$a_{21}$	$\emptyset$	$\emptyset$
$t_3$	$t_2$	0100	19:24	$\emptyset$	$a_{22}$	$\emptyset$	$\emptyset$
$t_4$	$t_3$	0001*	13:04	$\emptyset$	$\emptyset$	$\emptyset$	$c_2$
$t_5$	$t_4$	0101	19:25	$\emptyset$	$a_{22}$	$\emptyset$	$c_{21}$
$t_6$	$b_3$	0001*	15:05	$\emptyset$	$\emptyset$	$\emptyset$	$c_3$
$t_7$	$t_6$	0001	19:45	$\emptyset$	$\emptyset$	$\emptyset$	$c_{31}$
$t_8$	$b_1$	0000	20:15	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 2: An example of the update and delete procedures (conceptual tabular representation).

tail record is assigned a unique RID) are created and appended to the corresponding tail pages. For example, consider updating the column  $A$  of the record with the key  $k_2$  (referenced by the RID  $b_2$ ) in Table 2. The first tail record, referenced by the RID  $t_1$ , contains the original value of the updated column, i.e.,  $a_2$ , whereas implicit null values ( $\emptyset$ ) are preassigned for remaining unchanged columns. Taking snapshot of the original changed values becomes essential in order to ensure contention-free merging as discussed in Section 3.1. The second tail record contains the newly updated value for column  $A$ , namely,  $a_{21}$ , and again implicit special null values for the rest of the columns; a column that has never been updated does not even have to be materialized with special null values. However, for any subsequent updates, only one tail record is created, e.g., the tail record  $t_3$  is appended as a result of updating the column  $A$  from  $a_{21}$  to  $a_{22}$  for the record  $b_2$ .

In general, updates could either be cumulative or non-cumulative. The cumulative property implies that when creating a new tail record, the new record will contain the latest values for all of the updated columns thus far. For example, consider updating the column  $C$  for the record  $b_2$ . Since the column  $C$  of the record  $b_2$  is being updated for the first time, we first take a snapshot of its old value as captured by the tail record  $t_4$ . Now for the cumulative update, a new tail record is appended that repeats the previously updated column  $A$ , as demonstrated by the tail record  $t_5$ . If non-cumulative update approach was employed, then the tail record would consist of only the changed value for column  $C$  and not  $A$ . It is important to note that cumulation of updates can be reset at anytime. In the absence of cumulation, readers are simply forced to walk back the chain of recent versions to retrieve the latest values of all desired columns. Thus, cumulative update is an optimization that is intended to improve the read performance.

As part of the update routine, the embedded *Indirection* column (forward pointers) for base records is also updated to point to the newly created tail record. In our running example, the *Indirection* column of the record  $b_2$  points to the tail record  $t_5$ . Also after updating the column  $C$  of the record  $b_3$ , the *Indirection* column points to the latest version of  $b_3$ , which is given by  $t_7$ . Likewise, the *Indirection* column in the tail records are updated to point to the previous version of the record. It is important to note that the *Indirection* column of base records is the only column that requires an in-place update in our architecture. However, as discussed in our concurrency model (cf. Section 4.1), this is a special column that lends itself to latch-free concurrency protocol.

Furthermore, indexes always point to base records (i.e., base

RIDs), and they are never directly point to any tail records (i.e., tail RIDs) in order to avoid the index maintenance cost that arise in the absence of in-place update mechanism [33, 32]. Therefore, when a new version of a record is created (i.e., a new tail record), first, all indexes defined on unaffected columns do not have to be modified and, second, only the affected indexes are modified with the updated values, but they continue to point to base records and not the newly created tail records [33, 32]. Suppose there is an index defined on the column  $C$  (cf. Table 2). Now after modifying the record  $b_2$  from  $c_2$  to  $c_{21}$ , we add the new entry ( $c_{21}, b_2$ ) to the index on the column  $C$ .<sup>3</sup> Subsequently, when a reader looks up the value  $c_{21}$  from the index, it always arrives at the base record  $b_2$  initially, then the reader must determine the visible version of  $b_2$  (by following the indirection if necessary) and must check if the visible version has the value  $c_{21}$  for the column  $C$ , essentially re-evaluating the query predicates.

There are two other meta-data columns that are affected by the update procedure. The *Start Time* column for tail records simply holds the time at which the record was updated (an implicit end of the previous version). For example, the record  $t_7$  has a start time of 19:45, which also implies that the end time of the first version of the record  $b_3$ . The *Schema Encoding* column is a concise representation that shows which data columns have been updated thus far. For example, the *Schema Encoding* of the tail record  $t_7$  is set to “0100”, which implies that only the column  $A$  has been changed. To distinguish between whether a tail record is holding new values or it is the snapshot of old values, we add a flag to the *Schema Encoding* column, which is shown as an asterisk. For example, the tail record  $t_6$  stores the old value of the column  $A$ , which is why its *Schema Encoding* is set to “0100\*”. The *Schema Encoding* can also be maintained optionally for base records as part of the update process or it could be populated only during the merge process.

Notably, when there are multiple individual updates to the same record by the same transaction, then each update is written as a separate entry to tail pages. Each update results in a creation of a new tail record and only the final update becomes visible to other transactions. The prior entries are implicitly invalidated and skipped by readers. Also delete operation is simply translated into an update operation, in which all data columns are implicitly set to  $\emptyset$ , e.g., deleting the record  $b_1$  results in creating the tail record  $t_8$ . An alternative design for delete is to create a tail record that holds a complete snapshot of the latest version of the deleted record.

### 2.3.2 Insert Procedure

The final key operation is the insertion of new records. Conceptually, the table naturally grows by inserting new records to the end of the table (append-only mechanism). We rely on a simpler manifestation of our notion of tail pages followed by the transformation of tail pages into compressed, read-only base pages through a simplified merge process. In fact, one can even view our previously described update mechanism as a form of sparse insertion.

In our proposed insert design, we designate the end of the table as the insert range. An insert range is basically a pre-allocated range of base RIDs for accommodating future insertions. In practice, the insert range size (at least a million RIDs) is much larger than our range partitioning that is employed for update processing (i.e., up-

<sup>3</sup>Optionally the old value ( $c_2, b_2$ ) could be removed from the index; however, its removal may affect those queries that are using indexes to compute answers under snapshot semantics. Therefore, we advocate deferring the removal of changed values from indexes until the changed entries fall outside the snapshot of all relevant active queries.



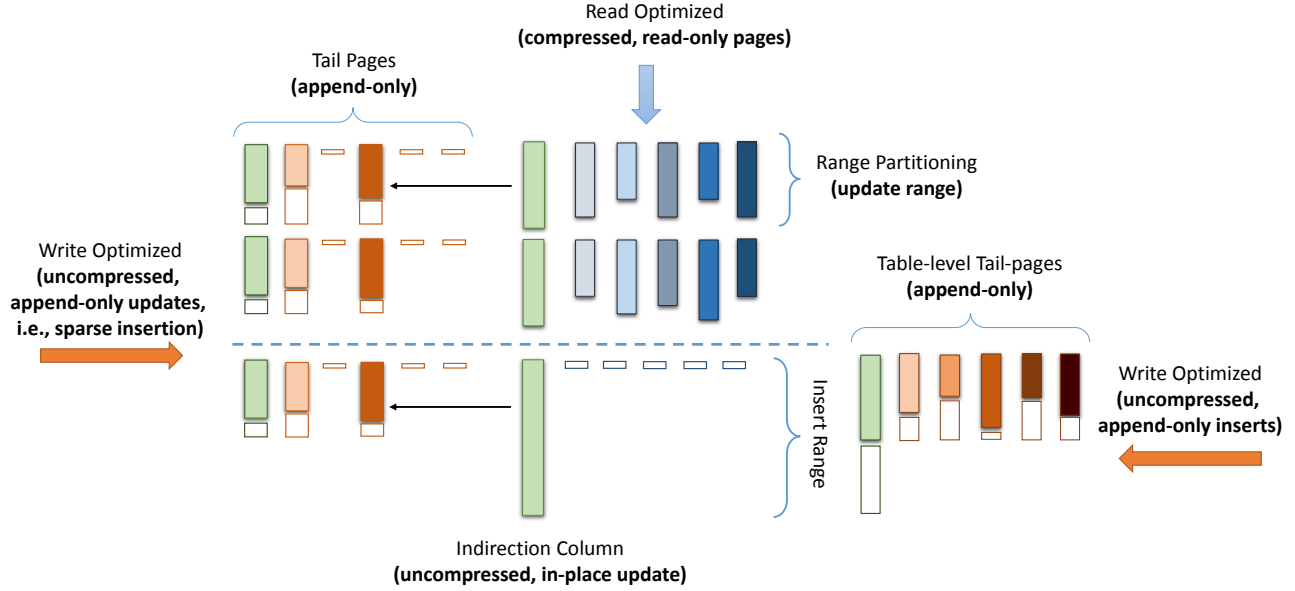


Figure 4: Append-only insertion of new records with concurrent updates (by employing tail pages).

RID	Indirection	Schema Encoding	Start Time	Key	A	B	C
Partitioned base records for the key range of $k_4$ to $k_6$							
$b_4$	$\perp$	0000	16:20	$k_4$	$a_4$	$b_4$	$c_4$
$b_5$	$\perp$	0000	17:21	$k_5$	$a_5$	$b_5$	$c_5$
$b_6$	$\perp$	0000	18:02	$k_6$	$a_6$	$b_6$	$c_6$
Insert range for the base record with the base RID range of $b_7$ to $b_9$							
$b_7$	$\perp$						
$b_8$	$t_{14}$						
$b_9$	$t_{16}$						
Table-level tail-pages for the base record with the base RID range of $b_7$ to $b_9$							
$tt_7$	$t_7$	0000	18:30	$k_7$	$a_7$	$b_7$	$c_7$
$tt_8$	$t_8$	0000	18:45	$k_8$	$a_8$	$b_8$	$c_8$
$tt_9$	$t_9$	0000	19:05	$k_9$	$a_9$	$b_9$	$c_9$
Partitioned tail records for the key range of $k_7$ to $k_9$							
$t_{13}$	$b_8$	0001*	18:45	$\emptyset$	$\emptyset$	$\emptyset$	$c_8$
$t_{14}$	$t_{13}$	0001	22:25	$\emptyset$	$\emptyset$	$\emptyset$	$c_{81}$
$t_{15}$	$b_3$	0100*	19:05	$\emptyset$	$a_9$	$\emptyset$	$\emptyset$
$t_{16}$	$t_{15}$	0100	22:45	$a_{91}$	$\emptyset$	$\emptyset$	$\emptyset$

Table 3: An example of insertion with concurrent updates (conceptual tabular representation).

date range).<sup>4</sup> For the insert range, we allocated a set of tail pages for appending new records, which we refer to them as “*table-level tail-pages*” even though structurally there is no difference between table-level tail-pages vs. regular tail pages. Figure 4 pictorially captures our insert design. In table-level tail-pages, we allocate tail pages for all columns (unlike for updates that was limited to only the updated columns) because the insert statement always provide a value for every column (even if it is an implicit null value for a nullable column).

Adding a new insert range consists of reserving a set of base RIDs (e.g., in the order of millions) and a set of tail RIDs; these two sets of RIDs are equal in size and aligned. Thus, the  $10^{th}$  base RID in the insert range corresponds to the  $10^{th}$  tail RID in the table-level tail-range (i.e., both ranges following the same insertion order). The alignment of RIDs allows implicit addressing for look-

ing up a record in the insert range. When a new record is about to be inserted to the table, the new record receives a reserved base RID in the insert range and the corresponding tail RID in the table-level tail-range. If insert range is full, then a new insert range is created. But the key guiding principle for insertion is to satisfy the *stability property* of the base pages (i.e., read-only) with the exception of the *Indirection* column that is updated in-place. Therefore, the insertion procedure simply consists of acquiring base and tail RIDs, insert the actual record to table-level tail-pages, and setting the *Indirection* column in the base record to null. Alternatively, the *Indirection* column could be set to null when allocating pages for the insert range.

An example of insertion is illustrated in Table 3. The insert range is shown as  $b_7$  to  $b_9$ , and the table-level tail-range is shown as  $tt_7$  to  $tt_9$ . The first inserted record is  $(k_7, a_7, b_7, c_7)$  with the key  $k_7$  that is assigned  $b_7$  as its base RID and  $tt_7$  as its tail RID. The only column allocated for base records is the *Indirection* column, which is initially set to null ( $\perp$ ). The actual values for the meta-data and data columns are appended to the table-level tail-pages at the position given by the tail RID  $tt_7$ . In the same spirit, the records with  $b_8$  and  $b_9$  are also appended to the insert range. Now if a recently inserted record is updated, then the update follows the same path as explained earlier (cf. Section 2.3.1). Suppose the record  $b_8$  is updated by modifying the value of its *C* column from  $c_8$  to  $c_{81}$ . The update simply results in acquiring a new tail RID in the regular tail pages (as before) and appending only the updated column followed by updating the *Indirection* column in-place. This is demonstrated by appending the tail record  $t_{14}$  to the corresponding tail pages and setting the *Indirection* column of the record  $b_8$  to  $t_{14}$ .

### 3. REAL-TIME STORAGE ADAPTION

To ensure a near optimal storage layout, outdated base pages are merged lazily with their corresponding tail pages in order to preserve the efficiency of analytical query processing. Recall that the base pages are read-only and compressed (read optimized) while

<sup>4</sup>Each table may have more than one insert range to support a higher degree of concurrency if the workload is insert intensive.

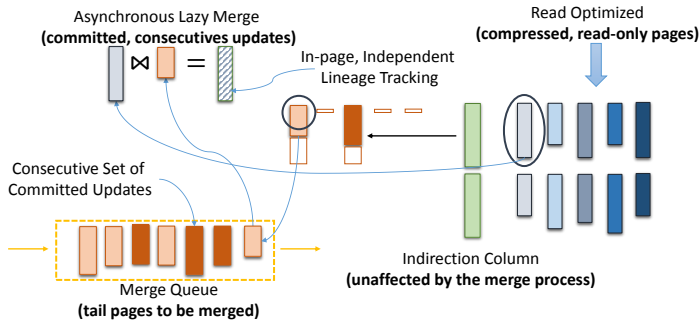


Figure 5: Lazily, independently merging of tail & base pages.

the tail pages are uncompressed<sup>5</sup> that grow using a strictly append-only technique (write optimized). Therefore, it is necessary to transform the recent committed updates (accumulated in tail pages) that are write optimized into read optimized form. A distinguishing feature of our lineage-based architecture is to introduce a contention-free merging process that is carried out completely in the background without interfering with foreground transactions. Furthermore, the contention-free merging procedure is applied only to the updated columns of the affected update ranges. There is even no dependency among columns during the merge; thus, the different columns of the same record can be merged completely independent of each other at different point in time. The merge process is conceptually depicted in Figure 5, in which writer threads (i.e., update transactions) place candidate tail pages to be merged into the merge queue while the merge thread continuously takes pages from the queue and processes them.

### 3.1 Contention-free, Relaxed Merge

In L-Store, we abide to one *main design principle* for ensuring contention-free processing that is “*always operating on stable data*”. The inputs to the merge process are (1) a set of base pages (committed base records) that are read-only,<sup>6</sup> thus, stable data and (2) a set of consecutive committed tail records in tail pages,<sup>7</sup> thus, also stable data. The output of the merge process (that is also relaxed) is a set of newly consolidated base pages (also referred to as merged pages) that are read-only, compressed, and almost up-to-date, thus, stable data. To decouple users’ transactions (writers) from the merge process, we also ensure that the write path of the ongoing transactions does not overlap with the write path of the merge process. Writers append new uncommitted tail records to tail pages (but as stated before uncommitted records do not participate in the merge), and writers perform in-place update of the *Indirection* column within base records to point to the latest version of the updated records in tail pages (but the *Indirection* column is not modified by the merge process), whereas the write path of the merge process consists of creating only a new set of read-only base pages.

<sup>5</sup>Even though compression techniques such as local and global dictionaries can be employed in tail pages, but these directions are outside the scope of the current work.

<sup>6</sup>The *Indirection* column is the only column that undergoes in-place update that also never participates in the merge process.

<sup>7</sup>Note that not every committed update has to be applied as the merge process is relaxed, and the merge eventually process all committed tail records.

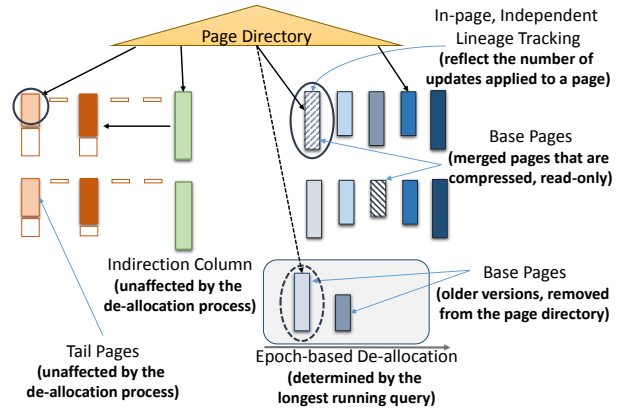


Figure 6: Epoch-based, contention-free page de-allocation.

#### 3.1.1 Merge Algorithm

The details of the merge algorithm, conceptually resembling the standard left-outer join, consists of (1) identifying a set of committed tail records in tail pages; (2) loading the corresponding outdated base pages; (3) consolidating the base and tail pages; (4) updating the page directory; and (5) de-allocating the outdated base pages.

**Step 1: Identify committed tail records in tail pages:** Select a set of consecutive fully committed tail records (or pages) since the last merge within each update range.

**Step 2: Load the corresponding outdated base pages:** For a selected set of committed tail records, load the corresponding outdated base pages for the given update range (limit the load to only outdated columns). This step can further be optimized by avoiding to load sub-ranges of records that have not yet changed since the last merge. No latching is required when loading the base pages.

**Step 3: Consolidate the base and tail pages:** For every updated column, the merge process will read  $n$  outdated base pages and applies a set of recent committed updates from the tail pages and writes out  $m$  new pages.<sup>8</sup> First the Base RID column of the committed tail pages (from Step 1) are scanned in reverse order to find the list of the latest version of every updated record since the last merge (a temporary hashtable may be used to keep track whether the latest version of a record is seen or not). Subsequently, applying the latest tail records in a reverse order to the base records until an update to every record in the base range is seen or the list is exhausted (skip any intermediate versions for which a newer update exists in the selected tail records). If a latest tail record indicates the deletion of the record, then the deleted record will be included in the consolidated records.<sup>9</sup> Any compression algorithm (e.g., dictionary encoding) can be applied on the consolidated pages (on column basis) followed by writing the compressed pages into newly

<sup>8</sup>At most up to one merged page per column could be left under-utilized for a range of records after the merge process. To further reduce the underutilized merged pages, one may define finer range partitioning for updates (e.g.,  $2^{12}$  records), but operate merges at coarser granularity (e.g.,  $2^{16}$  records). This will provide the benefit of locality of access for readers given smaller range size of  $2^{12}$ , yet it provides a better space utilization and compression for newly created merge pages when larger ranges are chosen.

<sup>9</sup>Alternatively, if all the deleted values are also stored in tail records, then it is sufficient to fill all data columns with the special null value  $\emptyset$  for deleted records in the final merged pages. However, we would still need to preserve the *Indirection* column of deleted records in order to provide access to the earlier versions of deleted records.

RID	Indirection	Schema Encoding	Start Time	Last Updated Time	Key	A	B	C
Partitioned base records for the key range of $k_1$ to $k_3$ ; Tail-page Sequence Number (TPS) = 0								
$b_1$	$t_8$	0000	10:02		$k_1$	$a_1$	$b_1$	$c_1$
$b_2$	$t_5$	0101	13:04		$k_2$	$a_2$	$b_2$	$c_2$
$b_3$	$t_7$	0001	15:05		$k_3$	$a_3$	$b_3$	$c_3$
Relevant tail records (below TPS $\leq t_7$ high-watermark) for the key range of $k_1$ to $k_3$								
$t_5$	$t_4$	0101	19:25		$\emptyset$	$a_{22}$	$\emptyset$	$c_{21}$
$t_7$	$t_6$	0001	19:45		$\emptyset$	$\emptyset$	$\emptyset$	$c_{31}$
Resulting merged records for the key range of $k_1$ to $k_3$ ; TPS = $t_7$								
$b_1$	$t_8$	0000	10:02	10:02	$k_1$	$a_1$	$b_1$	$c_1$
$b_2$	$t_5$	0101	13:04	19:25	$k_2$	$a_{22}$	$b_2$	$c_{21}$
$b_3$	$t_7$	0001	15:05	19:45	$k_3$	$a_3$	$b_3$	$c_{31}$

Table 4: An example of the relaxed and almost up-to-date merge procedure (conceptual tabular representation).

created pages. Moreover, the old Start Time column is remained intact during the merge process because this column is needed to hold the original insertion time of the record.<sup>10</sup> Therefore, to keep track of the time for the consolidated records, the Last Updated Time column is populated to store the Start Time of the applied tail records. The Schema Encoding column may also be populated during the merge to reflect all the columns that have been changed for each record.

**Step 4: Update the page directory:** The pointers in the page directory are updated to point to the newly created merged pages. Essentially this is the only foreground action taken by the merge process, which is simply to swap and update pointers in the page directory – an index structure that is updated rarely only when new pages are allocated.

**Step 5: De-allocate the outdated base pages:** The outdated base pages are de-allocated once the current readers are drained naturally via an epoch-based approach. The epoch is defined as a time window, in which the outdated base pages must be kept around as long as there is an active query that started before the merge process. Pointers to the outdated base pages are kept in a queue to be re-claimed at the end of the query-driven epoch-window. The pointer swapping and the page de-allocation are illustrated in Figure 6. ■

An example of our merge process is shown in Table 4 based on our earlier update example, in which we apply the first seven tail records (denoted by  $t_1$  to  $t_7$ ) to their corresponding base pages. The resulting merged pages are shown, where the affected records are highlighted. Note that only the updated columns are affected by the merge process (and the Indirection column is not affected). Furthermore, not all updates are needed to be applied, only the latest version of every updated record needs to be consolidated while the other entries are simply discarded. In our example, only the tail records  $t_5$  and  $t_7$  participated in the merge, and the rest were discarded.

It is important to note that merging table-level tail-pages with base pages in the insert range follows a similar process as above with a few simplification. First, the consolidation process is rather trivial because tail records in the table-level tail-range follows the exact same insertion order in the insert range (a trivial join-like operation). Also the insert range does not actually have any value except for the Indirection column (which does not even participate in the merge itself). Thus, the merge process is essentially reading a set of consecutive committed tail records and compressing them to create a set of newly merged pages. Another simplification is

<sup>10</sup>The Start Time column is also highly compressible column with a negligible space overhead to maintain it.

that after the merged pages are created and the page directory is updated, then the old table-level tail-pages can be discarded permanently after all the active queries that started prior to the merge process are terminated. In contrast, the regular tail pages survive after the merge in order to enable answering historic queries and to avoid interfering with update transactions. All base records that have been merged with their table-level tail-pages are considered to be outside the insert range.

We further strengthen our *data stability* condition for bringing base pages up-to-date. Earlier we stated that the merge only operates on a set of committed consecutive tail records, but no condition was imposed on the base records. Now we strengthen this condition by requiring that the base records must also fall outside the insert range before becoming a candidate for merging the recent updates.

### 3.1.2 Merge Analysis

A key distinguishing feature of our lineage-based storage architecture is to allow contention-free merging of tail and base pages without interfering with concurrent transactions. To formalize our merge process, we prove that merge operates only on stable data without any information loss and that the merge does not limit users' transactions to access and/or modify the data that is being merged.

**Lemma 1** Merge operates strictly on stable data.

**PROOF.** Recall that by construction, we enforced that merge “always operate on stable data”. The inputs to the merge process are (1) a set of base pages consisting of committed base records that are read-only and outside the insert range, thus, stable data and (2) a set of consecutive committed tail records in tail pages, thus, also stable data. The output of the merge process is a set of newly merged pages that are read-only, thus, stable data as well. Hence, the merge process strictly takes as inputs stable data and produces stable data as well. □

**Lemma 2** Merge safely discards outdated base pages without violating any query's snapshot.

**PROOF.** In order to support snapshot isolation semantics and time travel queries, we need to ensure that earlier versions of records that participate in the merge process are retained. Since we never perform in-place updates and each update is transformed into appending a new version of the record to tail pages, then as long as tail pages are not removed, we can ensure that we have access to every updated version. But recall that outdated base pages are de-allocated using our proposed epoch-based approach after being merged. Also note that base pages contain the original values of when a record was first created. Therefore, any original values that later were updated must be stored before discarding outdated base pages after a merge is taken place. In another words, we must ensure that outdated base pages are discarded safely.

As a result, the two fundamental criteria, namely, relaxing the merge (i.e. constructing an almost up-to-date snapshot) and operating on stable data, are not sufficient to ensure the *safety property* of the merge. The last missing piece that enables safety of the merge is accomplished by taking a snapshot of the original values when a column is being updated for the first time (as described in Section 2.3). In other words, we have further strengthened our *data stability* criterion by ensuring even stability in the committed history. Hence, outdated base pages can be safely discarded without any information loss, namely, the merge process is safe. □

**Theorem 1** The merge process and users' transactions do not contend for base and tail pages or the resulting merged pages, namely, the merge process is contention-free.



PROOF. As part of ensuring contention-free merge, we have already shown that merge operates on stable data (proven by Lemma 1) and that there is no information loss as a result of the merge process (proven by Lemma 2). Next we prove that the write path of the merge process does not overlap with the write path of users’ transactions (i.e., writers). Recall that writers append new uncommitted tail records to tail pages (but as stated before uncommitted records do not participate in the merge), and writers perform in-place update of the *Indirection* column within base records to point to the latest version of the updated records in tail pages (but the *Indirection* column is not modified by the merge process), whereas the write path of the merge process consists of creating only a new set of read-only merged pages and eventually discarding the outdated base pages safely.

Therefore, we must show that safely discarding base pages does not interfere with users’ transactions. In particular, as explained in Lemma 2, if the original values were not written to tail records at the time of the update, then during the merge process, we were forced to store them somewhere or encounter information loss. It is not even clear where would be the optimal location for storing the original values. A simple minded approach of just adding them to tail pages would have broken the linear order of changes to records such that the older values would have appeared after the newer values, and it would have interfered with the ongoing update transactions. But, more importantly, the need to store the old values at any location would have implied that during the merge process multiple coordinated actions were required to ensure consistency across modification to isolated locations; hence, breaking the contention-free property of the merge. Therefore, by storing the original updated values at the time of update, we trivially eliminate all the potential contention during the merge process in order to safely discarding outdated base pages.

As a result, users’ transactions are completely decoupled from the merge process, and users’ transactions and the merge process do not contend over base, tail, or merged pages.  $\square$

When analyzing the performance of our merge algorithm, we observe that in the worst case, data from all updated columns for a given update range is read and written back, but it is a cost that is amortized over many updates, a key strength of our lineage-based storage architecture. In general, if updates are spread over a range of records (even if skewed), then the data for the entire range has to be read and written; however, when updates are strictly localized, then additional optimization can be applied to further prune the set of records read.

However, it is important to note that L-Store’s objective has been to introduce a contention-free merge procedure without the interference with the concurrent transactions because contention is the most important deciding factor in the overall performance of the system especially as the size of the main memory continues to increase (arguably the entire transactional data can fit in the main memory today) and the storage-class memories (such as SSDs) replace the mechanical disks [8, 30]. Nevertheless, we point out that there are potential opportunities to further improve the merge process execution time by employing and studying more complex join algorithms for implementing the merge by operating directly on the compressed data without the need to decompress and compress the data (a complementary direction that is the outside the scope of the current work). Nevertheless, in our evaluation (Section 5), with a single asynchronous merge thread, we were able to cope with tens of concurrent writer threads, and we were able to process millions of updates per second when updating 40% of the columns on average.

Lastly, we would like to point out that there are also a num-

ber of potential opportunities to guide the merge process in order to further accelerate relaxed analytical queries (those queries that can tolerate slightly outdated snapshot) by implicitly constructing a slightly outdated but consistent snapshot of the data across the entire table during the merge. Currently, our proposed merge is already relaxed and brings base pages almost up-to-date in time. Now we suggest to further coordinate the merge such that every merge not only take a set of consecutive committed tail records, but also takes only those consecutive committed records before an agreed upon time  $t_i$ . Thus, after merging a range of records, we are ensured that only committed records before the time  $t_i$  is processed. Furthermore, we propose that every page also maintains its temporal lineage to remember the timestamp of the earliest committed records that have not been merged yet, where ideally its timestamp is after  $t_i$ . Any range of records that yet to be merged or has failed to bring base pages forward in time up to  $t_i$  can be manually brought forward to  $t_i$  as part of the normal query processing by consolidating tail pages. Periodically, the agreed upon merge time is advanced from  $t_i$  to  $t_{i+1}$ , and all subsequent merges are adjusted accordingly. However, exploiting the temporal lineage to speed up relaxed analytical queries (almost for free) is outside the scope of the current work.<sup>11</sup>

### 3.2 Maintaining Lineage

The lineage of each base page (and consequently merged pages) is maintained independently as a result of the merge process. The lineage information is instrumental to decouple the merge and update processing and to allow independent merging of the different columns of the same record at different point in time. The lineage information is captured using a rather simple and elegant concept, which we refer to as *tail-page sequence number (TPS)* in order to keep track of how many updated entries (i.e., tail records) from tail pages have been applied to their corresponding base pages after a completion of a merge. Original base pages always start with TPS set to 0, a value that is monotonically increasing after every merge. Again to ensure this monotonicity property, we stressed earlier that always a consecutive set of committed tail records are used in the merge process.

TPS is also used to interpret the indirection pointer (also a monotonically increasing value) by readers after the merge is taken place. Consider our running example in Table 4. After the first merge process, the newly merged pages have TPS set to 7, which implies that the first seven updates (tail records  $t_1$  to  $t_7$ ) in the tail pages have been applied to the merged pages. Consider the record  $b_2$  in the base pages that has an indirection value pointing to  $t_5$  (cf. Table 4), there are two possible interpretations. If the transaction is reading the base pages with TPS set to 0, then the 5<sup>th</sup> update has not yet reflected on the base page. Otherwise if the transaction is reading the base pages with TPS 7, then the update referenced by indirection value  $t_5$  has already been applied to the base pages as seen in Table 4. Notably, the *Indirection* column is updated only in-place (also a monotonically increasing value) by writers, while merging tail pages does not affect the indirection value.

More importantly, we can leverage the TPS concept to ensure read consistency of users’ transactions when the merge is performed

<sup>11</sup>Similar optimization for constructing an almost up-to-date and consistent snapshots was first introduced in [23], but it required to drain all active queries before the out-of-date snapshot could be advanced in time or it required maintaining multiple almost up-to-date snapshots simultaneously. Unlike the approach in [23], our proposed merge algorithm combined with the temporal lineage eliminates all contention with the ongoing queries such that the relaxed snapshot can be brought forward in time lazily and asynchronously.

RID	Indirection	Schema Encoding	Start Time	Last Updated Time	Key	A	B	C
Recently merged records for the key range of $k_1$ to $k_3$ ; TPS = $t_7$								
$b_1$	$t_8$	0000	10:02	10:02	$k_1$	$a_1$	$b_1$	$c_1$
$b_2$	$t_{12}$	0101	13:04	19:25	$k_2$	$a_{22}$	$b_2$	$c_{21}$
$b_3$	$t_{11}$	0001	15:05	19:45	$k_3$	$a_3$	$b_3$	$c_{31}$
Partitioned tail records for the key range of $k_1$ to $k_3$								
$t_1$	$b_2$	0100*	13:04		$\emptyset$	$a_2$	$\emptyset$	$\emptyset$
$t_2$	$t_1$	0100	19:21		$\emptyset$	$a_{21}$	$\emptyset$	$\emptyset$
$t_3$	$t_2$	0100	19:24		$\emptyset$	$a_{22}$	$\emptyset$	$\emptyset$
$t_4$	$t_3$	0001*	13:04		$\emptyset$	$\emptyset$	$\emptyset$	$c_2$
$t_5$	$t_4$	0101	19:25		$\emptyset$	$a_{22}$	$\emptyset$	$c_{21}$
$t_6$	$b_3$	0001*	15:05		$\emptyset$	$\emptyset$	$\emptyset$	$c_3$
$t_7$	$t_6$	0001	19:45		$\emptyset$	$\emptyset$	$\emptyset$	$c_{31}$
$t_8$	$b_1$	0000	20:15		$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$t_9$	$t_5$	0010*	13:04		$\emptyset$	$\emptyset$	$b_2$	$\emptyset$
$t_{10}$	$t_9$	0010	21:25		$\emptyset$	$\emptyset$	$b_{21}$	$\emptyset$
$t_{11}$	$t_7$	0001	21:30		$\emptyset$	$\emptyset$	$\emptyset$	$c_{32}$
$t_{12}$	$t_{10}$	0110	21:55		$\emptyset$	$a_{23}$	$b_{21}$	$\emptyset$

Table 5: An example of the indirection interpretation and lineage tracking (conceptual tabular representation).

lazily and independently for the different columns of the same records. Therefore, when the merge of columns is decoupled, each merge occurs independently and at a different point in time. Consequently, not all base pages are brought forward in time simultaneously. Additionally, even if the merge occurs for all columns simultaneously, it is still possible that a reader reads base pages for the column A before the merge (or during the merge before the page directory is updated) while the same reader reads the column C after the merge; thus, reading a set of inconsistent base and merged pages.

**Lemma 3** *An inconsistent read with concurrent merge is always detectable.*

PROOF. Since each base page independently tracks its lineage, i.e., its TPC counter; therefore, TPS can be used to verify the read consistency. In particular, for a range of records, all read base pages must have an identical TPS counter; otherwise, the read will be inconsistent. Hence, an inconsistent read across different columns of the same record is always detectable.  $\square$

**Theorem 2** *Constructing consistent snapshots with concurrent merge is always possible.*

PROOF. As proved in Lemma 3, the read inconsistency is always detectable. Furthermore, once a read inconsistency is encountered, then each page is simply brought to the desired query snapshot independently by examining its TPS and the indirection value and consulting the corresponding tail pages using the logic outlined earlier. Hence, consistent reads by constructing consistent snapshots across different columns of the same record is always possible.  $\square$

TPS, or an alternative but similar counter conceptually, could be used as a high-water mark for resetting the cumulative updates as well. Continuing with our running scenario, in which we have the original base pages with the TPS 0 (as shown in Table 4), the merged pages the with TPS 7 (as shown in Table 5). For simplicity, we assume the cumulation was also reset after the  $7^{th}$  tail record. For the record  $b_2$ , we see that the indirection pointer is  $t_{12}$ , for which we know that the cumulative update has been reset after the  $7^{th}$  update. This means that the tail record  $t_{12}$  does not carry updates that were accumulated between tail records 1 to 7. Suppose that the record was updated four times, where the update entries in the tail pages are  $3^{rd}$ ,  $5^{th}$ ,  $10^{th}$ , and  $12^{th}$  tail records. The tail record  $t_5$  is a cumulative and carries the updated values from the

RID	Indirection	Schema Encoding	Start Time	A	C
Merged, committed tail records for the key range of $k_1$ to $k_3$					
$t_1$	$b_2$	0100*	13:04	$a_2$	$\emptyset$
$t_2$	$t_1$	0100	19:21	$a_{21}$	$\emptyset$
$t_3$	$t_2$	0100	19:24	$a_{22}$	$\emptyset$
$t_4$	$t_3$	0001*	13:04	$\emptyset$	$c_2$
$t_5$	$t_4$	0101	19:25	$a_{22}$	$c_{21}$
$t_6$	$b_3$	0001*	15:05	$\emptyset$	$c_3$
$t_7$	$t_6$	0001	19:45	$\emptyset$	$c_{31}$
Ordered, Inlined, Compressed committed tail records for the key range of $k_1$ to $k_3$					
$c_1$	$b_2$	0101	13:04,19:21,19:24,19:25	$a_2,a_{21},a_{22}$	$c_2,c_{21},c_{22}$
$c_2$	$b_3$	0001	15:05,19:45	$\emptyset$	$c_3,c_{31}$

Table 6: An example of compressing merged tail pages (conceptual tabular representation).

tail record  $t_3$ . However, the tail record  $t_{10}$  is not cumulative (reset occurred at the  $8^{th}$  update), whereas the tail record  $t_{12}$  is cumulative, but carries updates only from the tail record  $t_{10}$  and not from  $t_5$  and  $t_3$ . Suppose that a transaction is reading the base pages with the TPS 0, then to reconstruct the full version of the record  $b_2$ , it must read both the tail records  $t_5$  and  $t_{12}$  (while skipping  $3^{rd}$  and  $10^{th}$ ). But if a transaction is reading from the merged pages with the TPS 7, then it is sufficient to only read the tail record  $t_{12}$  to fully reconstruct the record because the  $3^{rd}$  and  $5^{th}$  updates have already been applied to the merged pages.

### 3.3 Compressing Historic Data

For historic tail pages, namely, the committed and subsequently merged tail pages, we introduce a contention-free compression scheme to substantially reduce storage footprint and improving access patterns for historic queries (or time travel queries). Periodically, for a range of records, we compress only a set of merged tail pages (across all updated columns) that fall outside the oldest query snapshot in order to avoid clashing with the readers/writers of non-historic data.

The key benefit of our compression scheme, which takes as input a set of tail pages for all updated columns for a given range of records (stable data) and outputs a set of newly compressed tail pages (in columnar form), are the following key properties (as demonstrated in Table 6). First, the compressed tail records are re-ordered according to the base RID order; hence improving the locality of access. Second, for each record, and within each column, the different versions are stored inline and contiguously. The version inlining avoid the need to repeatedly store unchanged values due to cumulative updates, but, more importantly, it enables delta compression among the different versions of the records to further reduce the space overhead. Also collapsing the different versions of the same record into a single tail record eliminates the need for back pointers that are needed for referencing the previous versions. Thus, the inline versions are tightly packed and ordered temporally as shown in Table 6.

The compressed tail pages are read-only and are used exclusively for historic queries. These new pages can be isolated and pushed down lower in the storage hierarchy since they are by definition colder and not accessed as frequently. Lastly, the page directory is updated by swapping the pointers for the old tail pages to point to the newly created compressed tail pages. Notably, considering that the access frequency is much lower for historic tail pages compared to the access frequency of non-historic tail pages, any locking- or non-locking-based approaches (such as the epoch-based approach discussed in Section 3.1) can be employed without noticeably affecting the overall system performance.

### 3.4 Record Partitioning Trade-offs

When choosing the range of records for partitioning (i.e., update range) there are several dimensions that needs to be examined. An important observation is that regardless of the range size, recent updates to tail pages will be memory resident and no random disk I/O is required; this trend is supported by continued increase in the size of main memory and the fact that the entire OLTP database is expected to fit in the main memory [8, 18].

In our evaluation, we studied in-depth the impact of the range size, and we observed that the key deciding factor is the frequency at which the merges are processed. How frequent a merge is initiated is proportional to how many tail records are accumulated before the merge process is triggered. We further experimentally observed that the update range sizes in order of  $2^{12}$  to  $2^{16}$  exhibit a superior overall performance vs. data fragmentation depending on the workload update distribution. Because for a smaller update range size, we may have many corresponding half-filled tail pages, but as the range size increases, the cost of half-filled tail pages are amortized over a much larger set of records.<sup>12</sup> Furthermore, the range size effects the clustering of updates in tail pages, the larger the range size, then it is more likely that cache misses occur when scanning the recent update that are not merged yet. Again, considering that recent cache sizes are in order of tens of megabytes, the choice of any range value between  $2^{12}$  to  $2^{16}$  is further supported. As noted before, one may choose a finer range partitioning for handling updates (i.e., update range), e.g.,  $2^{12}$ , to improve locality of access while choosing coarser virtual range sizes when performing merges, essentially forcing the merge to take-in as input a set of consecutive update ranges that have been updated, e.g., choosing  $2^4$  consecutive  $2^{12}$  ranges in order to merge  $2^{12} \times 2^4 = 2^{16}$  records.

For example, suppose that scan operations (even if there are concurrent scans operations) may access 2 columns, assume each column is  $2^3$  bytes long. We further assume that the merge can keep up, namely, even for  $2^{16}$  update range size, the number of tail records yet to be merged is less than  $2^{16}$  (as shown in Section 5, such merging rate can be achieved while executing up to 16 concurrent update transactions). The overall scan footprint (combining both base pages and tail pages) is approximately  $2^{16} \times 2^3 \times 2 \times 2 = 2^{21}$  (2 MB), which certainly fits in today's processor cache (in our evaluation, we used Intel Xeon E5-2430 processor, which has 15 MB cache size). Thus, even as scanning base records, if one is forced to perform random lookup within a range of  $2^{16}$  tail records, the number of cache misses are limited compared to when the range size was beyond the cache capacity.

Another criteria for selecting an effective update range size is the need for RID allocation. In L-Store, upon the first update to a range of records (e.g.,  $2^{12}$  to  $2^{16}$  range), we pre-allocate  $2^{16}$  unused RIDs for referencing its corresponding tail pages. Tail RIDs are special in a sense they are not added to indexes and no unique constraint is applied on them. Once the tail RID range is fully used, then either a new unused RID range is allocated or an existing underutilized tail RID range can be re-assigned (partially used RID range must satisfy TPS monotonicity requirement). Furthermore, in order to avoid overlapping the base and tail RIDs, one could assign tail RIDs in the reverse order starting from  $2^{64}$ ; therefore, tail RIDs will be monotonically decreasing, and the TPS logic must be reversed accordingly. The benefit of reverse assignment is that while scanning page directory for base pages, there is no need to first read

and later skip tail page entries (read optimization).

## 4. FAST TRANSACTIONAL CAPABILITIES

In order to support concurrent transactions (where each transaction may consists of many statements), any database engine must provide necessary functionalities to ensure the correctness of concurrent reads and writes of the shared data. Furthermore, transaction logging is required in order to recover the system from crash and media failure.

### 4.1 Simplified Concurrency & Recovery

Our lineage-based storage architecture consists of read-only base pages (that are not modified) and append-only updates to tail pages (which are not modified once written). When a record is updated, no logging is required for base pages (because they are read-only), but the modified tail pages requires redo logging. Again, since we eliminate any in-place update for tail pages, no undo log is required. Upon a crash, the redo log for tail pages are replayed, and for any uncommitted transactions (or partial rollback), the tail record is marked as invalid (e.g., tombstone), but the space is not reclaimed until the compression phase.

The one exception to above rule for logging and recovery is the *Indirection* column, which is updated in-place. There are two possible recovery options: (1) one can rely on standard undo-redo log for the *Indirection* column only or (2) one can simply rebuilt the *Indirection* column upon crash. The former option can further be optimized based on the realization that tail pages undergo strictly redo policy and aborted transactions do not physically remove the aborted tail records as they are only marked as tombstones. Therefore, it is acceptable for the *Indirection* column to continue pointing to tombstones, and from the tombstones finding the latest committed values. As a result, even for the *Indirection* column only the redo log is necessary. For the latter recovery option, as discussed earlier, to speedup the merge process, we materialize the *Base RID* column in tail records that can be used to populate the *Indirection* column after the crash. Alternatively, even without materializing an additional RID column, one can follow backpointers in the *Indirection* column of tail records to fetch the base RID because the very first tail record always points back to the original base record.

The merge process is idempotent because operates strictly on committed data and repeated execution of the merge always produce the exact same results given a set of base pages, their corresponding tail pages, and a merge threshold that dictates how many consecutive committed tail records to be used in the merge process. Therefore, only operational logging is required for the merge process. Also updating the entries in the page directory upon completion of the merge process simply requires standard index logging (both undo-redo logs). If crash occurs during the merge, simply the partial merge results can be ignored and the merge can be restarted. Similarly, compressing the historic tail-pages is idempotent and requires only operational logging and restart on crash.

In terms of concurrency protocol for transaction processing, any existing protocols can be leveraged because L-Store's primary focus is the storage architecture. In particular, we relied on the recently proposed optimistic concurrency model introduced in [30] that supports full ACID properties for multi-statement transactions, and we also employed the speculative reads proposed in [17]. But in terms of low-level latching, our lineage-based storage has set of unique benefits, namely, readers do not have to latch the read-only base pages or fully committed tail pages. Also there is no need to latch partially committed tail pages when accessing committed records. More importantly, writers never modify base pages (except the *Indirection* column) nor the fully committed tail pages, so no latch-

<sup>12</sup>To reduce space under-utilization, tail pages could be smaller than base pages, for instance, tail pages could be 4 KB while base pages are 32 KB or larger.

ing is required for stable pages. The *Indirection* column is at most 8-byte long; therefore, writers can simply rely on atomic compare-and-swap (CAS) operators to avoid latching the page. Also as part of the merge process, no latching of tail and base pages are required because they are not modified. The only latching requirement for the merge is updating the page directory to point to the newly created merged pages. Therefore, every affected page in the page directory are latched one at a time to perform the pointer swap or alternatively atomic CAS operator is employed for each entry (pointer swap) in the page directory. Alternatively, the page directory can be implemented using latch-free index structures such as Bw-Tree [20].

## 4.2 Miscellaneous: Write-ahead Logging

In the columnar storage, writing the log record becomes even a more expensive operation because a single record update or insert spans multiple columns that reside on different pages. As a result, when writing the log record, all affected pages due to updates must be latched with an exclusive access. While the pages are latched, the necessary changes are made to the pages, the log record is written, the log sequence number (LSN) is acquired, and the *pageLSN* for every page is updated to hold the LSN of the latest update [22]. Subsequently, all the exclusive latches are released. Holding an exclusive latch is essential otherwise the page may end up in an inconsistent state.

Consider two transactions attempting to update two different records on the same page. The first transaction  $t_1$  receives LSN  $l_1$  while the second transaction  $t_2$  receives LSN  $l_2$ . Suppose log records are written without holding the latch on the page that needs to be updated. In a concurrent system, it is possible that  $t_2$  first gets to update the page, then updates the *pageLSN* to  $l_2$  while  $t_1$  has yet to update the page. At this point, the page has entered an inconsistent state because the *pageLSN* indicates that all updates up to  $l_2$  are applied to the page while  $t_1$  has not made any changes. Now if  $t_1$  arrives and make the desired updates, then it is still unclear how  $t_1$  can announce that its changes are applied because if it attempts to update the *pageLSN* to  $l_1$ , then the page again will be in an inconsistent state because both updates from  $t_1$  and  $t_2$  have been applied, but the *pageLSN* only indicates that update from  $t_1$  has been applied. At this point, the page is dirty with an incorrect *pageLSN* again. Suppose the bufferpool stealing policy is now exercised, and the dirty page with the *pageLSN*  $l_1$  (but having updates from both  $t_1$  and  $t_2$ ) is written to disk. Now consider the scenario, in which the page is flushed and the transaction  $t_1$  commits, but while  $t_2$  is still active the database crashes. Now after the crash, the page in question appears to be clean because the latest change on the page denoted by  $l_1$  has committed; hence, dirty uncommitted changes from  $t_2$  will not be discovered, and the database will be inconsistent. An alternative scenario is when the page is flushed to disk (again due to bufferpool stealing policy) after  $t_2$  updates the *pageLSN*, but before  $t_1$  make any changes to the page. Thus, the inconsistent data becomes persistent. Suppose  $t_1$  request the page again, and the page is brought back to memory, and  $t_1$  updates the page content (and leaves the *pageLSN* to remain at  $l_2$ ), and  $t_1$  commits. Now if the database crashes, the persistent page on disk has the *pageLSN*  $l_2$  and both  $t_1$  and  $t_2$  have committed, but changes from  $t_2$  are only reflected on disk and updates from  $t_1$  are lost. However, the crash recovery protocol sees that the page is up-to-date and clean and no further action is taken; hence, the database will remain in an inconsistent state.

Although holding an exclusive latch while updating a page solves the above mentioned inconsistencies, but holding an exclusive latch for such a prolonged duration will substantially deteriorates the

overall performance of the system. In L-Store, the base pages are not updated in-place, and tail pages follow a strict append-only policy. But even with an append-only policy if the write-ahead logging is employed, then it may deteriorate the transaction throughput due to exclusive latches. Similar challenges also arise even when updating a single column, namely, the *Indirection* column (but without the need of undo log). To address the WAL challenges for columnar storage, we introduce an Ownership Relaying (OR) protocol to substantially reduce the need for holding exclusive latches in order to correctly update the *pageLSN*. The key idea behind OR protocol is to have all writers to hold a compatible shared latch instead (not exclusive latches) while only one transaction (with the highest LSN) is selected as the owner of the page and responsible for updating the *pageLSN* and promoting its shared latch to an exclusive one. Other transactions do not need to update the *pageLSN*; however, before releasing the shared latches, they must ensure that the page has an owner. Therefore, we propose that each data page to have an *ownerLSN* in addition to the *pageLSN*.<sup>13</sup>

For multi-threaded in-place update of the *Indirection* column, our proposed OR protocol will be as follows. All writers (i.e., update transactions) acquire a shared latch on the *Indirection* column before making any changes. After the latch is granted, the *Indirection* column is updated in-place, the redo log record is written, and the LSN is acquired. If the *ownerLSN* of the page is larger than the writer's LSN, then the shared latch is released. However, if the *ownerLSN* is smaller than the writer's LSN, then the writer updates the *ownerLSN* (using atomic CAS operator) and promotes its shared latch to an exclusive latch (a conditional promotion), and checks if it is still the owner while waiting otherwise the latch is released. Once the exclusive latch is granted, the writer will update the *pageLSN* and release the latch.

Therefore, if there are 100 concurrent writers, then only one writer will get an exclusive latch on behalf of all the writers, and the *pageLSN* is updated once for every 100 concurrent writers. Since the writer will never release its shared latch as long as it is deemed owner, then we will never flush (nor persist) a page when the page content and the *pageLSN* are not yet consistent. Also to ensure that the page is flushed eventually to avoid starvation, periodically a page is forced to drain all its current writers and update *pageLSN* before accepting any new writers (similar to a checkpointing procedure). This can be implemented by ensuring that at most  $\theta_s$  shared latches are granted between any two consecutive flushes. Therefore, once the threshold  $\theta_s$  is exceeded, then no new shared latches are granted for writers and the page is forced to drain its writers and be flushed subsequently. Recall that readers do not need to hold any shared latch when reading the *Indirection* column because all changes to the page are done using atomic CAS operator; thus, the forced flushing policy does not affect the readers.

For multi-threaded append-only updates (sparse inserts) or inserts to tail pages, we propose slightly more complicated variation of our OR protocol. We further assume that every page is pre-allocated with a set of fixed size slots (the number of available slot is maintained for each page) to accommodate the updates. The writer first acquires a shared latch on the *Indirection* column in the tail page followed by acquiring a tail RID for appending the updates (or a new record). For appending a tail record, each writer acquires a shared latch for each updated column individually. The latches are acquired in the order in which columns appear in the table schema (from left to right) and held until the update is completed. Once the shared latches are granted, the writer writes the

<sup>13</sup>The *ownerLSN* does not have to be materialized on the page itself and could be maintained as a meta-data in an external data structure.

new value in the pre-allocated slots determined by the assigned tail RID. After append is completed, the redo log record is written, and the LSN is acquired. For every page, if the writer's LSN is the highest LSN that the page has seen so far ( $LSN \geq ownerLSN$ ), then the writer becomes the page owner and updates the *ownerLSN* using atomic CAS operator; otherwise the shared latch is released. For each page that the writer has the ownership, the writer promotes its shared latch to an exclusive latch, and checks if it is still the owner while waiting for the latch; otherwise, it releases the latch. While holding the exclusive latch, the writer will update the *pageLSN*, and optionally will compact slots and pre-allocate new slots for future updates as needed. Notably even if the writer with the ownership is aborted, the tail entries will not be removed; thus, the write will continue to update the *pageLSN* accordingly.

Handling the starvation problem for page flushing is much simpler for tail pages, and it requires no intervention. As soon as a tail page is full, then naturally it will have no more writers, so it can be flushed without the need to introduce any forced flushing policy.

## 5. EXPERIMENTAL EVALUATION

In order to study the impact of high-throughput transaction processing in the presence of long-running analytical queries, we carried out a comprehensive set of experiments. These experiments were run using an existing micro benchmark proposed in [17, 30], for the sake of a fair comparison and evaluation. This benchmark allows us to study different storage architectures by narrowing down the impact of concurrency with respect to the database active set by adjusting the degree of contention between readers and writers.

### 5.1 Experimental Setting

We evaluate the performance of various aspects of our real-time OLTP and OLAP system. Our experiments were conducted on a two-socket Intel Xeon E5-2430 @ 2.20 GHz server that has 6 cores per socket with hyper-threading enabled (providing a total of 24 hardware threads). The system has 64 GB of memory and 15 MB of L3 cache per socket. We implemented a complete working prototype of L-Store and compared it against two different techniques, (i) In-place Update + History and (ii) Delta + Blocking Merge, which are described subsequently. The prototype was implemented in Java (using JDK 1.7).<sup>14</sup> Our primary focus here is to simultaneously evaluate read and write throughputs of these systems under various transactional workloads concurrently executed with long-running analytical queries, which is the key characteristic of any real-time OLTP and OLAP system.

Our employed micro benchmark [17, 30] consists of three key types of workloads: (1) low contention, where the database active set is 10M records; medium contention, where the active set is 100K records; and high contention, where the active set is 10K records. It is important to note that the database size is not limited to the active set and can be much larger (millions or billions of records). Similar to [17, 30], we consider two classes of transactions: read-only transactions (executed under snapshot isolation semantics) that scan up to 10% of the data (to model TPC-H style analytical queries) and short update transactions (to model TPC-C and TPC-E transactions), in which each transaction consists of 8 read and 2 write statements (executed under committed read semantics) over a table schema with 10 columns. In addition, we vary the ratio of read/writes in these update transactions to model different customer scenarios with different read/write degrees. By default, transactional throughput of these schemes are evaluated while running (at least) one scan thread and one merge thread to create the

real-time OLTP and OLAP scenario. Unless stated explicitly, the percentage of reads and writes in the transactional workload is fixed at 80% and 20%, respectively. On average 40% of all columns are updated by the writers. Lastly, the page size is set to 32 KB for both base and tail pages because a larger page size often results in a higher compression ratio suitable for analytical workloads [13].

Next we describe the two techniques that are compared with L-Store. We point out the primary features of these techniques and describe it with respect to L-Store. For fairness, across all techniques, we have maintained columnar storage, maintained a single primary index for fast point lookup, and employed the embedded-indirection column to efficiently access the older/newer versions of the records. Additionally, logging has been turned off for all systems as logging could easily become the main bottleneck (unless sophisticated logging mechanisms such as group commits and/or enterprise-grade SSDs are employed). In the In-place Update + History technique, we are required to write both redo-undo logs for all updates while for L-Store and Delta + Blocking Merge only redo log is needed due to their append-only scheme.

**In-place Update + History (IUH):** A prominent storage organization is to append old versions of records to a history table and only retain the most recent version in the main table, updating it in-place. An example of a commercial system that has implemented this table organization is the Oracle Flashback Archive [24]; thus, our In-place Update + History is inspired by such table organization that avoids having multiple copies and representations of the data. However, due to the nature of the in-place update approach, each page requires standard shared and exclusive latches that are often found in major commercial database systems. In addition to the page latching requirement, if a transaction aborts, then the update to the page in the main table is undone, and the previous record is restored. Scans are performed by constructing a consistent snapshots, namely, if records in the main table are invisible with respect to query's read time, then the older versions of the records are fetched from the history table by following the indirection column. In our implementation of In-place Update + History, we also ignored other major costs of in-place update over the compressed data, in which the new value may not fit in-place due to compression and requires costly page splits or shifting data within the page as part of update transactions. We further optimized the history table to include only the updated columns as opposed to inserting all columns naively.

**Delta + Blocking Merge (DBM):** This technique is inspired by HANA [15], where it consists of a main store and a delta store, and undergoes a periodic merging and consolidation of the main and delta stores. However, the periodic merging requires the draining of all active transactions before the merge begins and after the merge ends. Although the resulting contention of the merge appears to be limited to only the boundary of the merge for a short duration, the number of merges and the frequency at which this merge occurs has a substantial impact on the overall performance. We optimized the delta store implementation to be columnar and included only the updated columns [27]. Additionally, we applied our range partitioning scheme to the delta store by dedicating a separate delta store for each range of records to further reduce the cost of merge operation in presence of data skew. The partitioning allow us to avoid reading and writing the unchanged portion of the main store.

### 5.2 Experimental Results

In what follows, we present our comprehensive evaluation results in order to compare and study our proposed L-Store with respect to state-of-the-art approaches.

**Scalability under contention:** In this experiment, we show how

<sup>14</sup>Our prototype is implemented over the Apache Spark libraries [4].



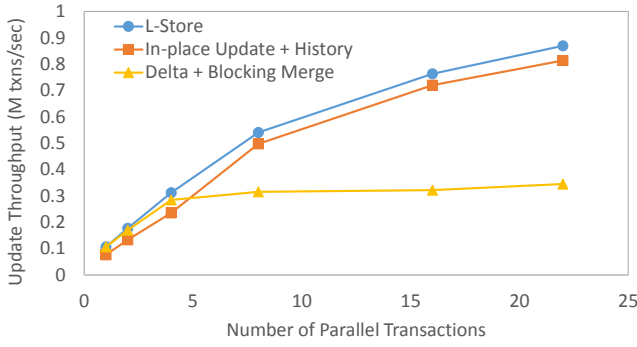


Figure 7: Scalability under low contention.

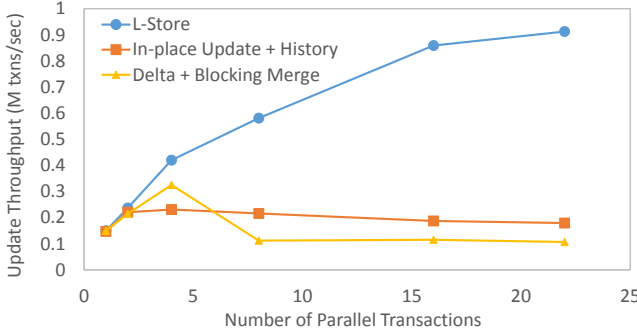


Figure 8: Scalability under medium contention.

transaction throughput scales as we increase the number of update transactions, in which each update transaction is assigned to one thread. For the scalability experiment, we fix the number of reads to 8 and writes to 2 for each transaction against a table with active set of  $N = 10$  million rows. Figure 7 plots the transaction throughput (y-axis) and the number of update threads (x-axis). Under low contention, the throughput for L-Store and In-place Update + History scales almost linearly before data is spread across the two NUMA nodes. The Delta + Blocking Merge approach however does not scale beyond a small number of threads due to the draining of active transaction before/after of each merge process, which brings down the transaction throughput noticeably. With increasing number of threads, the number of merges and the draining of active transactions become more frequent, which reduces the transaction throughput significantly. The In-place Update + History approach has lower throughput compared to L-Store due to the exclusive latches held for data pages that block the readers attempting to read from the same pages. The presence of a single history table also results in reduced locality for reads and more cache misses.

In addition, we study impact of increasing the degree of contention by varying the size of the database active set. For a fixed degree of contention, we vary the number of parallel update transactions from 1 to 22. For both medium contention (Figure 8) and high contention (Figure 9), we observe that L-Store consistently outperforms the In-place Update + History and Delta + Blocking Merge techniques as the number of parallel transactions is increased. For medium contention, we observed a speedup of up to  $5.09\times$  compared to the In-place Update + History technique and up to  $8.54\times$  compared to the Delta + Blocking Merge technique. Similarly for high contention, we observed up to  $40.56\times$  and  $14.51\times$  speedup with respect to the In-place Update + History and Delta + Blocking Merge techniques, respectively. The greater performance gap is attributed to the fact that in In-place Update + History, latching contention on the page is increased that is altogether elimi-

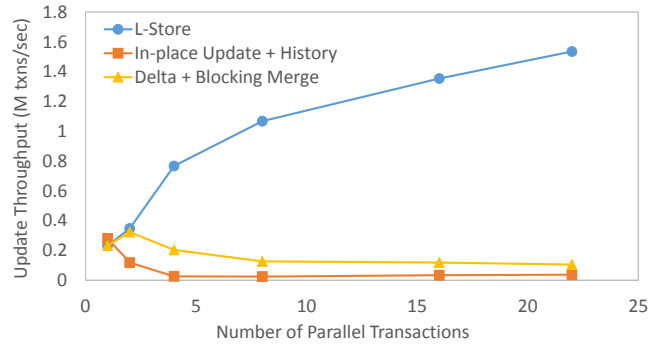


Figure 9: Scalability under high contention.

nated in L-Store. In Delta + Blocking Merge, since the active set is smaller, and all updates are concentrated to smaller regions, the merging frequency is increased, which proportionally reduces the overall throughput due to the constant draining of all active transactions. Finally, due to the smaller active set sizes in the medium- and high-contention workloads, the cache misses are also reduced as the cache-hit ratio increases. As a consequence, the transaction throughput also increases proportionately.

**Scan Scalability:** Scan performance is an important metric for real-time OLTP and OLAP systems because it is the basic building block for assembling complex ad-hoc queries. We measure the scan performance of L-Store by computing the SUM aggregation on a column that is continuously been updated by the concurrent update transactions. Thus, the goal of this experiment is to determine whether the merge can keep up with high-throughput OLTP workloads. As such, this scenario captures the worst-case scan performance because it may be necessary for the scan thread to search for the latest values in the merged page or tail pages when the merge cannot cope with the update throughput. For columns which do not get updated, the latest values are available in the base page itself, as described before. In this experiment (Figure 10), we study the single-threaded scan performance with one dedicated merge thread. We vary the number of tail records ( $M$ ) that are processed per merge (x-axis) and observe the corresponding scan execution time (y-axis) while keeping the range partitioning fixed at  $64K$  records. We repeat this experiment by fixing the number of update threads to 4 and 16, respectively. In general, we observe that as we increase  $M$ , the scan execution time decreases. The main reasoning behind this observation is that the scan thread visits tail pages for the latest values less often because the merge is able to keep up. However, for the smaller values of  $M$ , the merge is triggered more frequently and cannot be sustained. Additionally, the overall cost of the merge is increased because the cost of merge is amortized over fewer tail records while still reading the entire range of  $64K$  base records. Notably, if we delay the merge by accumulating too many tail records, then there is slight deterioration in performance. Therefore, it is important to balance the merge frequency vs. the amortization cost of the merge for the optimal performance, which based on our evaluation, it is when  $M$  is set to around 50% of the range size.

We also compare the single-threaded scan performance (for low contention and  $4K$  range size) of L-Store with the other two techniques in the presence of 16 concurrent update threads (See Table 7). Our technique outperforms the In-place Update + History and Delta + Blocking Merge techniques by 14.28% and 36.84%, respectively. It is important to note that smaller update range sizes, namely, assigning separate tail pages for each  $4K$  base records instead of  $64K$  base records, increases the overall scan performance

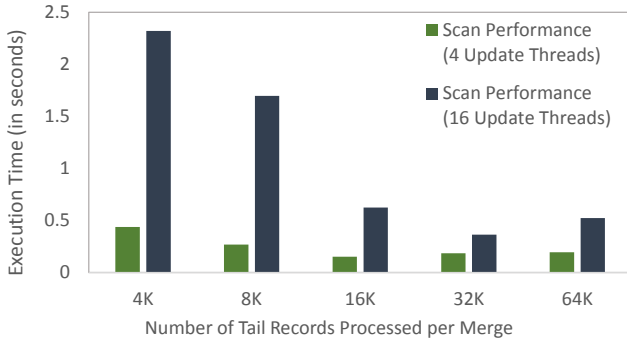


Figure 10: Scan Performance.

	L-Store	IUH	DBM
Scan Performance (in secs.)	0.24	0.28	0.38

Table 7: Scan Performance for different systems.

by improving the locality of access within tail pages. Therefore, as mentioned previously, it is beneficial to apply (virtual) fine-grained partitioning over base records (e.g., 4K records) to handle updates in order to improve locality of access within tail pages while applying (virtual) coarser-grained partitioning (e.g., 64K records) when performing the merge in order to reduce the space fragmentation in the resulting merged pages.

**Impact of varying the workload read/write ratio:** Short update transactions update only a few records in the database while performing reads for a majority of the time. A typical transactional workload comprises of 80% read statements and 20% writes [17]. However, our goal is to explore the entire spectrum from a read-intensive workload (read/write ratio 10:0) to a write-intensive workload (read/write ratio 0:10) while fixing the number of update threads to 16 [30]. Figure 11 shows transaction throughput (y-axis) as the ratio of read-only transactions varies in the workload (x-axis) with low contention. As expected, the performance of all the schemes increases as we increase the ratio of reads in the transactions because contention is a function of writes. As we have more writes in the workload, In-place Update + History technique suffers from increased contention as acquiring read latches conflict with the exclusive latches resulting in an extended wait time. The performance of the Delta + Blocking Merge technique also exacerbates since increasing the number of writes increases the number of merges performed. This brings down the performance further due to frequent halt of the system while draining active transactions. However, note that the gap between all of the schemes is the least when the workload consists of 100% reads. In summary, the speedup obtained with respect to In-place Update + History is up to  $1.45\times$  and up to  $5.78\times$  with respect to Delta + Blocking Merge technique. Note, even for 100% read, In-place Update + History continues to pay the cost of acquiring read latches on each page.

We repeat the same experiment but restrict the database active set size to 100K rows (Figure 12). L-Store significantly outperforms the other techniques across all workloads while varying the read/write ratio. But the performance gap is similar with respect to the low contention scenario when there are no update statements in the workload. The speedup obtained compared to In-place Update + History and Delta + Blocking Merge techniques is up to  $4.19\times$  and up to  $6.34\times$  respectively.

**Impact of long-read transactions:** As mentioned previously, it is not uncommon to have long-running read-only transactions in real-time OLTP and OLAP systems. These analytical queries

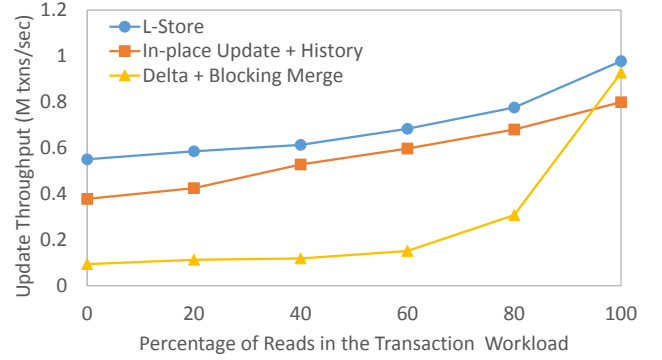


Figure 11: Impact of varying the read/write ratio of short update transactions (Low Contention).

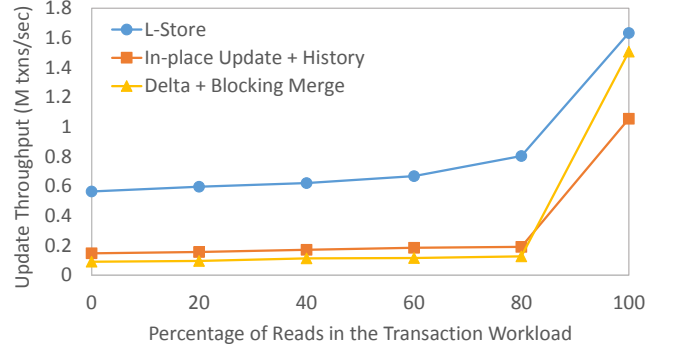


Figure 12: Impact of varying the read/write ratio of short update transactions (Medium Contention).

touch a substantial part of the database compared to the short update transactions, and the main goal is to reduce the interference between OLTP and OLAP workloads. In this experiment, we investigate the performance of the different schemes in the presence of these long-running read-only transactions, which on an average touch 10% of the base table. We fix the number of concurrent active transactions to 17 while increasing the number of concurrent read-only transactions from 1 to 16 (the short transactions simultaneously vary from 16 to 1). We also allocated a single merge thread for L-Store and Delta + Blocking Merge. Figure 13 represents the scenario for a low contention workload, while Figure 14 represents the scenario for medium contention. We observe that for both low and medium contention, there is an increase in throughput for both long-read transactions and short update transactions when the number of threads are increased. Moreover, the performance of read-only transaction increases for the medium contention scenario for all the techniques as the updates are restricted to a small portion of the database resulting in a higher read throughput. In other words, majority of the read-only transactions touch portions of the database in which updates do not take place resulting in higher throughput. For read-only transactions, our technique outperforms Delta + Blocking Merge up to  $1.97\times$  and  $2.37\times$  for low and medium contention workloads, respectively. For short update transactions, we outperform In-place Update + History and Delta + Blocking Merge by at most  $5.37\times$  and  $7.91\times$ , respectively, for medium-contention workload. In the earlier experiments, we had demonstrated that L-Store outperforms other leading approaches for update-intensive workloads, and in this experiment, we further strengthen our claim that L-Store substantially outperforms the leading approaches in the mixed OLTP and OLAP workload as well, the latter is due to our novel contention-free merging that does not interfere with the OLTP portion of the workload.

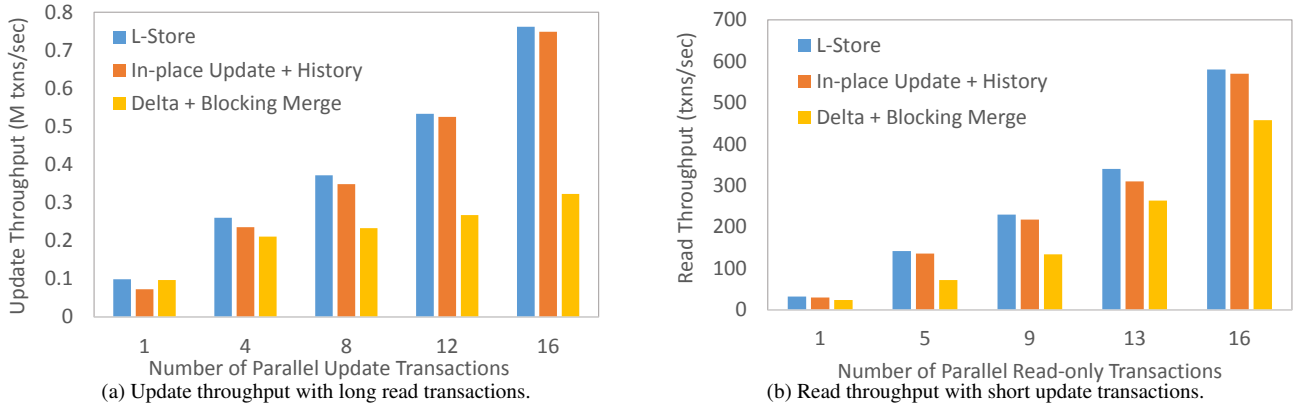


Figure 13: Throughput with long read-only transactions (Low Contention).

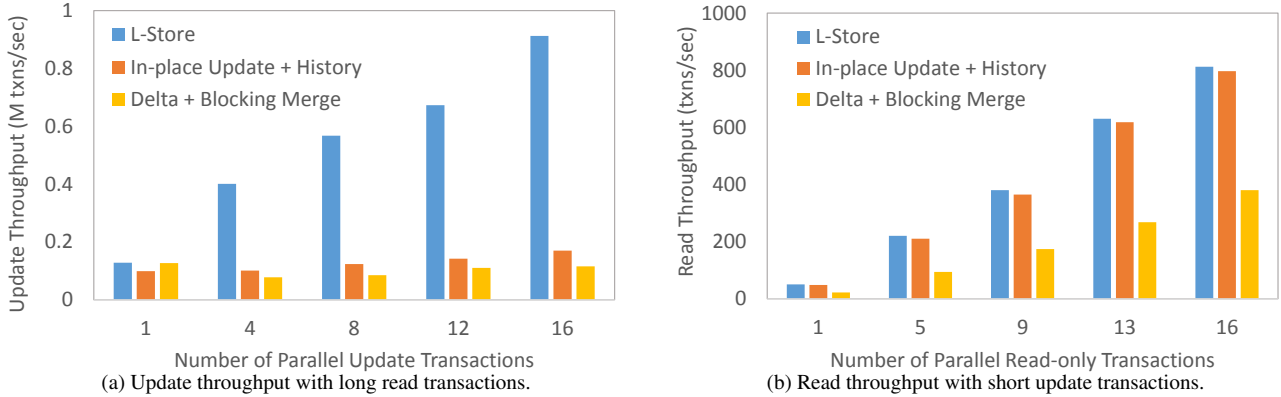


Figure 14: Throughput with long read-only transactions (Medium Contention).

## 6. RELATED WORK

In recent years, we have witnessed the development of many in-memory engines optimized for OLTP workloads either as research prototypes such as HyPer [13] and ES2 [6] or for commercial use such as Microsoft Hekaton [8], Oracle Database In-Memory [16], VoltDB [36], and HANA [15, 27]. Most of these systems are designed to keep the data in row format and in the main memory to increase the OLTP performance. In contrast, to optimize the OLAP workloads, columnar format is preferred. The early examples of these engines are C-Store [34] and MonetDB [5]. Recently, major database vendors also started integrating columnar storage format into their existing database engines. SAP HANA [10] is designed to handle both OLTP and OLAP workloads by supporting in-memory columnar format. IBM DB2 BLU [29] introduces a novel compressed columnar storage that is memory-optimized (and not restricted to being only memory-resident) that substantially improves the execution of complex analytical workloads by operating directly on compressed data. Below, we summarize the key aspects of recent developments that also aim to support real-time OLTP and OLAP workloads on the same platform.

HyPer, a main-memory database system, guarantees the ACID properties of OLTP transactions and supports running OLAP queries on consistent snapshot [13]. The design of HyPer leverages OS-processor-controlled lazy copy-on-write mechanism enabling to create a consistent virtual memory snapshot. Unlike L-Store, HyPer requires running transactions serially and assumes partitionable workload. Elastic power-aware data-intensive cloud computing platform (epiC) was designed to provide scalable database services on cloud [6]. epiC is designed to handle both OLTP and

OLAP workloads [7]. However, unlike L-Store, the OLTP queries in ES2 are limited to basic get, put, and delete requests (without multi-statements transactional support). Furthermore, in ES2, it is possible that snapshot consistency is violated and the user is notified subsequently [6].

Microsoft SQL Server currently contains three types of engines: the classical SQL Server engine designed to process disk-based tables in row format, the Apollo engine designed to maintain the data in columnar format [19], and the Hekaton in-memory engine designed to improve OLTP workload performance [8, 18]. In contrary, our philosophy is to avoid cost of maintaining multiple engines and to introduce a unified architecture to realize real-time OLTP and OLAP capabilities. Noteworthy, Microsoft has also recently announced moving towards supporting real-time OLTP and OLAP capabilities starting 2016 [18]. However, to provide OLTP and OLAP support among loosely integrated engines, a rather complex procedure is required as expected [18]. In particular, data are forced to move between Hekaton (a row-based engine) and the columnar indexes (based on the Apollo engine) in a cyclic fashion by creating a number of foreground transactions (e.g., one large transaction touching a million records on average followed by thousands of smaller transactions) that ultimately ran concurrently and creates potential contention with the users' transactions [18]. In contrast, in L-Store, we rely on purely columnar storage (while no multiple copies of the data is maintained) and, more importantly, our consolidation is based on a novel contention-free merge process that is performed asynchronously and completely in the background, and the only foreground task is pointer swaps in the page directory to point to the newly created merged pages.

Oracle has recently announced a new product called Oracle Database In-Memory that offers dual-format option to support real-time OLTP and OLAP. In the Oracle architecture, data resides in both columnar and row formats [16], whereas, in L-Store, we strictly keep only one copy and one representation of data. Last but not least, HANA [15, 27] is also designed as a real-time OLTP and OLAP engine, most notably, we share the same philosophy governing HANA that aims to develop a generalized solution for unifying OLTP and OLAP as opposed to building specialized engines. However, what distinguishes our architecture from HANA is that we propose a holistic columnar storage without the need to distinguishing between a main store and a delta store. In addition, we propose a contention-free merge process, whereas in HANA, the merge process is forced to drain all active transactions at the beginning and end of the merge process [15], a contention that results in a noticeable slow down as demonstrated in our evaluation.

On a different front, database concurrency theory, an old age problem, has recently been revived by industry (e.g., [17, 8]) and academia (e.g., [12, 37, 30]) due to hardware trends (e.g., multi-cores and large main memory) and application requirements (e.g., the need for processing millions of transactions per second in targeted advertising and algorithmic trading [31]). Hekaton focuses primarily on optimistic concurrency by assuming that roll backs are inexpensive and conflicts are rare [17]. Hekaton avoids the use of a lock manager and relies on read validation to ensure repeatable reads, performs re-execution of all range queries to achieve serializability, and detects write-write conflicts by using CAS operator and aborting the second writer to avoid any blocking. Furthermore, our past work goes beyond Hekaton by providing both efficient (latch-free) pessimistic and optimistic models that co-exists peacefully [30]. Similar to the approach in Hekaton [17], our past work [30] also rejects the idea of tuning the concurrency model to only limited types of workloads such as partitionable workloads (a direction that is pursued by [12, 37]). Since L-Store's focus is to provide a general storage architecture, any concurrency models can be employed; in particular, we relied on the optimistic concurrency model proposed in [30] while supporting the speculative reads proposed in [17].

## 7. CONCLUSIONS

In this paper, we develop Lineage-based Data Store (L-Store) to realize real-time OLTP and OLAP processing within a single unified engine. The key features of L-Store can succinctly be summarized as follows. In L-Store, recent updates for a range of records are strictly appended and clustered in its corresponding tail pages to eliminate read/write contention, which essentially transforms costly point updates into an amortized, fast analytical-like update query. Furthermore, L-Store achieves (at most) 2-hop access to the latest version of any record through an effective embedded indirection layer. More importantly, we introduce a novel contention-free merging of only stable data in order to lazily and independently bring base pages (almost) up-to-date without blocking on-going and new transactions. Furthermore, every base page relies on independently tracking the lineage information in order to eliminate all coordination and recovery even when merging different columns of the same record independently. Lastly, a novel contention-free page de-allocation using epoch-based approach is introduced without interfering with ongoing transactions. In our evaluation, we demonstrate that L-Store outperforms In-place Update + History by factor of up to  $5.37\times$  for short update transactions while achieving slightly improved performance for scans. It also outperforms Delta + Blocking Merge by  $7.91\times$  for short update transactions and up to  $2.37\times$  for long-read analytical queries.

## 8. ACKNOWLEDGMENTS

We wish to thank C. Mohan, V. Raman, R. Barber, R. Sidle, A. Storm, X. Xue, I. Pandis, Y. Chang, and G. M. Lohman for many insightful discussions and invaluable feedback in the earlier stages of this work.

## 9. REFERENCES

- [1] Net losses: Estimating the global cost of cybercrime. Center for Strategic and International Studies, June 2014, 2014.
- [2] Buy, buy, baby: The rise of an electronic marketplace for online ads is reshaping the media business. BIA/Kelsey, September 13, 2014.
- [3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01*, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
- [4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 1383–1394, New York, NY, USA, 2015. ACM.
- [5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.
- [6] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. ES2: A cloud data storage system for supporting both OLTP and OLAP. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE '11*, pages 291–302, Washington, DC, USA, 2011. IEEE Computer Society.
- [7] C. Chen, G. Chen, D. Jiang, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Providing scalable database services on the cloud. In *Web Information Systems Engineering—WISE 2010*, pages 1–19. Springer, 2010.
- [8] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwillig. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13*, pages 1243–1254, New York, NY, USA, 2013. ACM.
- [9] O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
- [10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.
- [11] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.
- [12] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.
- [13] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th*

- International Conference on Data Engineering, ICDE'11*, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.
- [14] L. Kim. How many ads does google serve in a day? Business 2 Community, November 2, 2012.
  - [15] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *Proc. VLDB Endow.*, 5(1):61–72, Sept. 2011.
  - [16] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle database in-memory: A dual format in-memory database. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, ICDE '15*, pages 1253–1258, April 2015.
  - [17] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwillig. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.
  - [18] P.-A. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. Real-time analytical processing with SQL server. *Proc. VLDB Endow.*, 8(12):1740–1751, Aug. 2015.
  - [19] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server column store indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11*, pages 1177–1184, New York, NY, USA, 2011. ACM.
  - [20] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering, ICDE '13*, Washington, DC, USA, 2013. IEEE Computer Society.
  - [21] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. IBM solidDB: In-memory database optimized for extreme speed and availability. *IEEE Data Eng. Bull.*, 36(2):14–20, 2013.
  - [22] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.
  - [23] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, SIGMOD '92*, pages 124–133, New York, NY, USA, 1992. ACM.
  - [24] Oracle Total Recall/Flashback Data Archive.
  - [25] E. Pacheco. Press release: Mobile will grab 11.5by 2019. BIA/Kelsey, April 22, 2015.
  - [26] K. Paterson. Credit card issuer fraud management, report highlights. Mercator Advisory Group, December 2008, 2008.
  - [27] H. Plattner. The impact of columnar in-memory databases on enterprise systems: Implications of eliminating transaction-maintained aggregates. *Proc. VLDB Endow.*, 7(13):1722–1729, Aug. 2014.
  - [28] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, Aug. 2012.
  - [29] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, Aug. 2013.
  - [30] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross. Reducing database locking contention through multi-version concurrency. *Proc. VLDB Endow.*, 7(13):1331–1342, 2014.
  - [31] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow.*, 3(1-2):1525–1528, Sept. 2010.
  - [32] M. Sadoghi, K. Ross, M. Canim, and B. Bhattacharjee. Exploiting ssds in operational multiversion databases. *The VLDB Journal*, pages 1–22, 2015.
  - [33] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making updates disk-I/O friendly using SSDs. *Proc. VLDB Endow.*, 6(11):997–1008, Aug. 2013.
  - [34] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05*, pages 553–564. VLDB Endowment, 2005.
  - [35] M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering, ICDE '05*, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.
  - [36] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.
  - [37] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD Conference*, pages 1–12. ACM, 2012.